

# Oracle<sup>®</sup> Application Server

Developer's Guide: C++ CORBA Applications

Release 4.0.8.1

September 1999

Part No. A70039-01

**ORACLE<sup>®</sup>**

The programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of such applications if the programs are used for such purposes, and Oracle Corporation disclaims liability for any damages caused by such use of the programs.

The programs (which include both the software and documentation) contain proprietary information of Oracle Corporation; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the programs is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error free. Except as may be expressly permitted in your license agreement for these programs, no part of these programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Oracle Corporation.

If the programs are delivered to the U.S. Government or anyone licensing or using the programs on behalf of the U.S. Government, the following notice is applicable:

**Restricted Rights Notice** Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication, and disclosure of the programs, including documentation, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication, and disclosure of the programs shall be subject to the restrictions in FAR 52.227-19, Commercial Computer Software - Restricted Rights (June, 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

Oracle is a registered trademark, and the Oracle logo, NLS\*WorkBench, Pro\*COBOL, Pro\*FORTRAN, Pro\*Pascal, SQL\*Loader, SQL\*Module, SQL\*Net, SQL\*Plus, Oracle7, Oracle Server, Oracle Server Manager, Oracle Call Interface, Oracle7 Enterprise Backup Utility, Oracle TRACE, Oracle WebServer, Oracle Web Application Server, Oracle Application Server, Oracle Network Manager, Secure Network Services, Oracle Parallel Server, Advanced Replication Option, Oracle Data Query, Cooperative Server Technology, Oracle Toolkit, Oracle MultiProtocol Interchange, Oracle Names, Oracle Book, Pro\*C, and PL/SQL are trademarks or registered trademarks of Oracle Corporation. All other company or product names mentioned are used for identification purposes only and may be trademarks of their respective owners.

---

---

# Contents

<b>Preface</b> .....	vii
<b>1 Overview</b>	
<b>A Component Based Application Model</b> .....	1-1
<b>Terminology</b> .....	1-2
<b>Client View of the C++ CORBA Cartridge</b> .....	1-3
<b>Container Architecture</b> .....	1-4
Lifecycle .....	1-5
Transactions.....	1-5
Load Balancing.....	1-5
Security.....	1-6
Resource Pooling (Stateful/Stateless C++ Cartridges) .....	1-6
<b>Application Development: General CORBA vs. C++ CORBA Cartridge</b> .....	1-6
<b>Using C++ Cartridge in an N-tier Computing Model</b> .....	1-7
<b>2 Tutorial</b>	
<b>Files in the Tutorial</b> .....	2-1
<b>1. Writing the Cartridge IDL File</b> .....	2-2
<b>2. Creating the Implementation Object</b> .....	2-3
<b>3. Creating the Deployment Descriptor File</b> .....	2-5
<b>4. Deploying the Application</b> .....	2-6
<b>5. Creating the Client Program</b> .....	2-8
<b>6. Creating the Client Executable</b> .....	2-11
<b>7. Running the Client to Access the C++ Application</b> .....	2-12

---

### 3 Developing C++ Cartridges

<b>Cartridge Remote Interface</b> .....	3-1
<b>C++ Implementation Object</b> .....	3-2
oas::cpp::Object Base Class .....	3-3
oas::cpp::Context Object .....	3-4
<b>Logging</b> .....	3-4
Overview .....	3-4
Example .....	3-5
<b>Transactions</b> .....	3-7
Overview .....	3-7
Example .....	3-7
<b>Cartridge Environment</b> .....	3-8
Overview .....	3-9
Example .....	3-9
<b>Stateful and Stateless Cartridges</b> .....	3-10
Stateful Cartridges .....	3-11
Stateless Cartridges .....	3-12

### 4 Creating the Deployment Descriptor File

<b>Overview</b> .....	4-1
<b>Structure of the Deployment Descriptor File</b> .....	4-1
Application Section .....	4-2
Cartridge Section .....	4-3

### 5 Developing Clients for C++ Applications

<b>Overview</b> .....	5-1
<b>Client Side Object Request Broker (ORB)</b> .....	5-2
<b>Getting the Object Reference for a C++ Object</b> .....	5-2
The Naming Tree .....	5-2
Bootstrapping .....	5-4
<b>Using the C++ Cartridge</b> .....	5-6
Invoking Methods on the C++ object .....	5-6
Destroying the C++ object .....	5-6
<b>Security</b> .....	5-7

---

<b>Example.....</b>	<b>5-7</b>
---------------------	------------

## 6 Installing C++ Applications

<b>Deploying Applications .....</b>	<b>6-1</b>
Generating Stubs and Skeletons .....	6-2
Generating C++ Cartridge Factories.....	6-2
Creating the Shared Library.....	6-2
Installing the Application .....	6-4
<b>Creating Client Executables.....</b>	<b>6-4</b>
Generating Stubs of the Standard CosNaming Interface.....	6-4
Compiling and Linking the Client Program.....	6-5
<b>Using the Utilities.....</b>	<b>6-5</b>
The IDL C++ Compiler .....	6-5
The cppgen Utility .....	6-9
The cppinstaller Utility .....	6-9
<b>Location of Your Registered C++ Application .....</b>	<b>6-9</b>
<b>Reinstalling and Reloading Applications.....</b>	<b>6-10</b>
Reinstalling C++ Applications from the Command-Line .....	6-11
<b>Configuring C++ Applications on Remote Nodes .....</b>	<b>6-12</b>
<b>Debugging Applications .....</b>	<b>6-13</b>
The Logger Class .....	6-13
Log Files .....	6-13
Severity Levels .....	6-13
Logging Modes .....	6-14
Troubleshooting Tips .....	6-15

## 7 Reference

### Index

---

---

# Preface

## Audience

The C++ CORBA cartridge defines a component architecture for building distributed object-oriented business applications in the C++ programming language. This book is for people who develop applications using the C++ CORBA cartridge of Oracle Application Server Release 4.0. It addresses the development, deployment, and runtime aspects of an application development lifecycle.

## The Oracle Application Server Documentation Set

This table lists the Oracle Application Server documentation set.

Title of Book	Part No.
Oracle Application Server 4.0.8 Documentation Set	A66971-03
Oracle Application Server Overview and Glossary	A60115-03
Oracle Application Server Installation Guide for Sun SPARC Solaris 2.x	A58755-03
Oracle Application Server Installation Guide for Windows NT	A58756-03
Oracle Application Server Administration Guide	A60172-03
Oracle Application Server Security Guide	A60116-03
Oracle Application Server Performance and Tuning Guide	A60120-03
Oracle Application Server Developer's Guide: PL/SQL and ODBC Applications	A66958-02
Oracle Application Server Developer's Guide: JServlet Applications	A73043-01
Oracle Application Server Developer's Guide: LiveHTML and Perl Applications	A66960-02
Oracle Application Server Developer's Guide: EJB, ECO/Java and CORBA Applications	A69966-01

---

Title of Book	Part No.
Oracle Application Server Developer's Guide: C++ CORBA Applications	A70039-01
Oracle Application Server PL/SQL Web Toolkit Reference	A60123-03
Oracle Application Server PL/SQL Web Toolkit Quick Reference	A60119-03
Oracle Application Server JServlet Toolkit Reference	A73045-01
Oracle Application Server JServlet Toolkit Quick Reference	A73044-01
Oracle Application Server Cartridge Management Framework	A58703-03
Oracle Application Server 4.0.8.1 Release Notes	A66106-04

## Conventions

This table lists the typographical conventions used in this manual.

Convention	Example	Explanation
bold	<b>oas.h</b> <b>owsctl</b> <b>wrbcfg</b> <b>www.oracle.com</b>	Identifies file names, utilities, processes, and URLs
italics	<i>file1</i>	Identifies a variable in text; replace this place holder with a specific value or string.
angle brackets	<filename>	Identifies a variable in code; replace this place holder with a specific value or string.
courier	owsctl start wrb	Text to be entered exactly as it appears. Also used for functions.
square brackets	[-c string] [on off]	Identifies an optional item. Identifies a choice of optional items, each separated by a vertical bar ( ), any one option can be specified.
braces	{yes no}	Identifies a choice of mandatory items, each separated by a vertical bar ( ).
ellipses	n,...	Indicates that the preceding item can be repeated any number of times.

The term “Oracle Server” refers to the database server product from Oracle Corporation.



---

The term “**oracle**” refers to an executable or account by that name.

The term “*oracle*” refers to the owner of the Oracle software.

## Technical Support Information

Oracle Global Support can be reached at the following numbers:

- In the USA: **Telephone: 1.650.506.1500**
- In Europe: **Telephone: +44 1344 860160**
- In Asia-Pacific: **Telephone: +61. 3 9246 0400**

Please prepare the following information before you call, using this page as a check-list:

- ☐ your CSI number (if applicable) or full contact details, including any special project information
- ☐ the complete release numbers of the Oracle Application Server and associated products
- ☐ the operating system name and version number
- ☐ details of error codes and numbers and descriptions. Please write these down as they occur. They are critical in helping WWCS to quickly resolve your problem.
- ☐ a full description of the issue, including:
  - **What** - What happened? For example, the command used and its result.
  - **When** -When did it happen? For example, during peak system load, or after a certain command, or after an operating system upgrade.
  - **Where** -Where did it happen? For example, on a particular system or within a certain procedure or table.
  - **Extent** - What is the extent of the problem? For example, production system unavailable, or moderate impact but increasing with time, or minimal impact and stable.
- ☐ Keep copies of any trace files, core dumps, and redo log files recorded at or near the time of the incident. WWCS may need these to further investigate your problem. For a list of trace and log files, see “Configuration and Log Files” in the *Administration Guide*.

---

For installation-related problems, please have the following additional information available:

- ❑ listings of the contents of \$ORACLE\_HOME (Unix) or %ORACLE\_HOME% (NT) and any staging area, if used.
- ❑ installation logs (**install.log**, **sql.log**, **make.log**, and **os.log**) typically stored in the \$ORACLE\_HOME/orainst (Unix) or %ORACLE\_HOME%\orainst (NT) directory.

## Documentation Sales and Client Relations

In the United States:

- To order hardcopy documentation, call Documentation Sales: **1.800.252.0303**.
- For shipping inquiries, product exchanges, or returns, call Client Relations: **1.650.506.1500**.

In the United Kingdom:

- To order hardcopy documentation, call Oracle Direct Response: **+44 990 332200**.
- For shipping inquiries and upgrade requests, call Customer Relations: **+44 990 622300**.

---

## Reader's Comment Form

### **Oracle Application Server Developer's Guide: C++ CORBA Applications** **Part No. A70039-01**

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have suggestions for improvement, please indicate the topic, chapter, and page number below:

Please send your comments to:

Oracle Application Server Documentation Manager  
Oracle Corporation  
500 Oracle Parkway  
Redwood Shores, CA 94065

If you would like a reply, please provide your name, address, and telephone number below:

Thank you for helping us improve our documentation.

---

---

# Overview

The C++ CORBA cartridge defines a component architecture for building distributed object-oriented business applications in the C++ programming language. It addresses the development, deployment, and runtime aspects of application development. The C++ CORBA cartridge makes it easy for application developers to write applications. The cartridge runtime shields the developers from low-level details of transactions, multi-threading, resource pooling, etc.

## Contents

- [A Component Based Application Model](#)
- [Terminology](#)
- [Client View of the C++ CORBA Cartridge](#)
- [Container Architecture](#)
- [Application Development: General CORBA vs. C++ CORBA Cartridge](#)
- [Using C++ Cartridge in an N-tier Computing Model](#)

## A Component Based Application Model

Oracle Application Server supports a component-based application model in the form of the C++ CORBA cartridge applications, Enterprise CORBA Object (ECO) for Java applications, and Enterprise JavaBeans (EJB) applications.

The C++ CORBA cartridges are created and managed at runtime by the C++ CORBA cartridge container, which is provided by Oracle Application Server. The characteristics of the cartridge, such as timeout values, state, etc. are customized at the time of deploying the application. A consistent view of the cartridge is given to

the client regardless of how the C++ cartridge is implemented and what functions it provides to the client.

In the Oracle Application Server environment, a C++ application consists of one or more C++ cartridges. C++ cartridges typically provide the business logic in C++ applications. The cartridges define methods that clients can invoke to perform some operation.

---

---

**Note:** Throughout this Guide the terms C++ CORBA application and C++ CORBA cartridge have been used interchangeably with the terms C++ application and C++ cartridge respectively.

---

---

## Terminology

This section defines the different entities that exist within the C++ CORBA cartridge world. The rest of the Guide uses this terminology:

- **Container:** Container is the C++ CORBA cartridge's runtime. This container provides security, concurrency, transactions, and other services to the C++ object. The implementations of these services are transparent to the client. The container itself lives in the cartridge server process of Oracle Application Server.
- **C++ application and C++ cartridge:** C++ application and C++ cartridge are named entities in your Oracle Application Server installation. A C++ application contains one or more C++ cartridges. A C++ cartridge consists of the business logic code that you write. All cartridges belonging to an application are packaged as a single module (shared object/DLL).
- **C++ object:** The same meaning as a "CORBA object" defined by the OMG specifications. It is virtual entity capable of being located by an ORB and having client requests invoked on it. It is "virtual" because it does not really exist until made real by an implementation written in a programming language.

A C++ object is an instance of a C++ cartridge. Each C++ object has an object reference. A client uses this object reference to invoke methods on the C++ object. The C++ object lives inside a container from the time of its creation to the time of its destruction.

- **C++ implementation object:** The same meaning as a "servant" defined by the OMG specifications. A C++ implementation object incarnates C++ objects: they provide bodies, or implementations, for C++ objects. C++ implementation objects exist within the context of a server process.

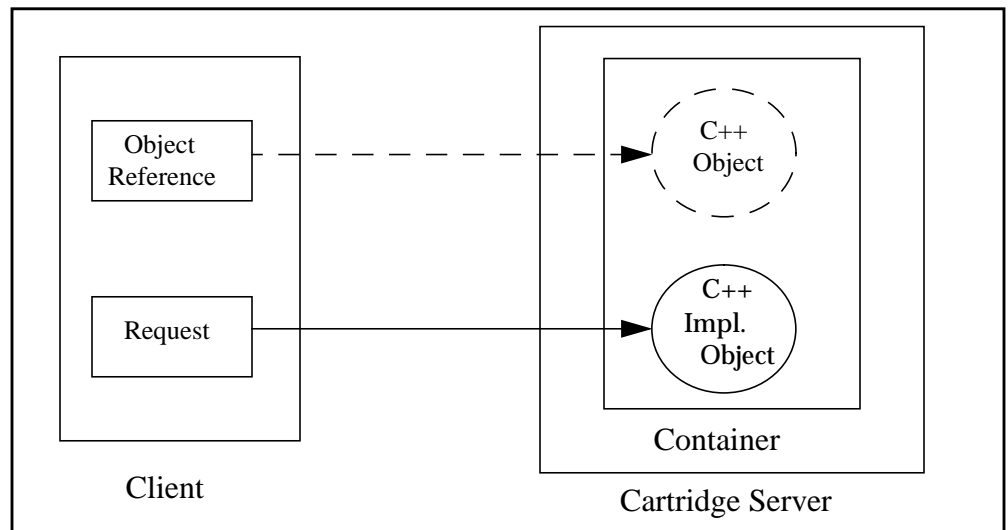
C++ implementation objects are instances of the C++ implementation class that you write.

See *Oracle Application Server Administration Guide* for more on cartridge servers and the architecture of Oracle Application Server.

## Client View of the C++ CORBA Cartridge

The container for the C++ CORBA cartridge provides security, concurrency, transactions, and other services to the C++ object. The implementations of these services are transparent to the client. [Figure 1-1](#) shows the client view of the C++ CORBA cartridge.

**Figure 1-1** Client view of the C++ CORBA cartridge



A client accesses the C++ cartridge using the standard CORBA CosNaming service as defined by the OMG (<http://www.omg.org>) COSS specification. The container binds the application and cartridge names into a CosNaming tree, which is used by the client to obtain the object reference to a C++ object.

The client uses the application name and the cartridge name to resolve the object reference for the C++ object. The following code snippet is an example of a client accessing a C++ cartridge. The client is accessing the cartridge Account in the Bank application:

```
CosNaming::Name name;
name.length(2);
name[0].id = CORBA::string_dup("Bank");
name[1].id = CORBA::string_dup("Account");

try
{
    CORBA::Object_ptr cpp = inc->resolve(name);
    if (CORBA::is_nil(cpp))
    {
        cerr << "Error resolving Bank/Account" << endl;
        exit(1);
    }
}
catch (CORBA::SystemException& se)
{
    cerr << "Error: " << se._repository_id() << endl;
    return 1;
}
```

The Oracle Application Server naming tree consists of C++ applications and cartridges deployed in Oracle Application Server. When you deploy new applications or cartridges in Oracle Application Server, they are added to this naming tree. The client's CosNaming name space is populated by the C++ CORBA cartridge container. Java clients can use JNDI to access a C++ cartridge.

See [Chapter 5, “Developing Clients for C++ Applications”](#) for details.

## Container Architecture

The C++ CORBA cartridge container is the runtime that manages the lifecycle (creation and destruction) of C++ objects. The container is implemented on a portable object adapter (POA) based C++ ORB. It provides the following services to C++ objects:

- [Lifecycle](#)
- [Transactions](#)
- [Load Balancing](#)
- [Security](#)
- [Resource Pooling \(Stateful/Stateless C++ Cartridges\)](#)



The container creates a Context object for every C++ object. The Context object makes the container services available to the C++ implementation object.

## Lifecycle

The container invokes lifecycle methods (Create and Remove) to the C++ implementation object. The following sequence of steps takes place when a client invokes business methods on a C++ object:

1. The client calls a `resolve()` or `secure_resolve()` on the root naming context of the naming tree to get the object reference of the C++ object.
2. The Oracle Application Server daemon creates a cartridge server process and instantiates the C++ object and the C++ implementation object.
3. The container creates a Context object and invokes lifecycle methods on the C++ implementation object.
4. The container returns the object reference of the C++ object to the client.
5. The client invokes business methods on the object reference. The container delegates these methods to the C++ implementation object.
6. The container destroys the C++ object when the client invokes a business method that calls the remove method on the Context object, or when the C++ object's timeout expires.

## Transactions

The C++ implementation object can use the Context object to access the UserTransaction object. The UserTransaction object allows the implementation object to begin, commit, and rollback transactions. The container supports distributed transactions across address spaces and across multiple Oracle databases. See [Chapter 3, “Developing C++ Cartridges”](#) for more on using transactions.

## Load Balancing

The container makes sure that there are enough threads to invoke methods on the C++ object. The container allocates resources based on two types of load balancing modes: priority based and min/max. You can configure your C++ application to select any of these two modes.

In the priority mode, the number of threads to execute methods on C++ objects will depend on the priority of the C++ application. In the min/max mode, as many threads as required by the application are created by the container. User threads

can also be created by the C++ object for dealing with the load balancing needs of the application. (We do not recommend this method.)

See *Oracle Application Server Performance and Tuning Guide* for more on how to configure your application to use priority based or min/max load balancing.

## Security

The container can protect C++ objects by requiring that clients authenticate themselves before it instantiates a C++ object. See [Chapter 5, “Developing Clients for C++ Applications”](#) and the *Oracle Application Server Security Guide*.

## Resource Pooling (Stateful/Stateless C++ Cartridges)

In a stateful C++ cartridge, the C++ implementation object contains conversational state, which is retained across method invocations. In a stateless C++ cartridge, the implementation object is shared. It does not contain any conversational state across method invocations. Thus, the instance can be used by any client.

Use stateful cartridges if you want the same state across multiple requests associated with a client. Use stateless cartridges if you do not want a conversational state associated with C++ objects, but want a light-weight application. Since stateless cartridges are shared, they are more efficient and scalable.

See [Chapter 3, “Developing C++ Cartridges”](#) for more on stateful and stateless C++ cartridges.

## Application Development: General CORBA vs. C++ CORBA Cartridge

General CORBA applications are different from C++ CORBA cartridge applications in terms of the development process. The CORBA application development process consists of the following six steps:

1. Determine your application’s objects and define their interfaces in IDL.
2. Compile your IDL definitions into C++ stubs and skeletons.
3. Declare and implement C++ servant classes that can incarnate your CORBA objects.
4. Write a server main program.
5. Compile and link your server implementation files with the generated stubs and skeletons to create your server executable.
6. Write, compile, and link your client code together with the generated stubs.

With the C++ CORBA cartridge you get the advantage of features provided by the Oracle Application Server that simplifies the development process. You donot have to write a server program, as specified in step 4. The container provides you with the services that a server would provide in a general CORBA application.

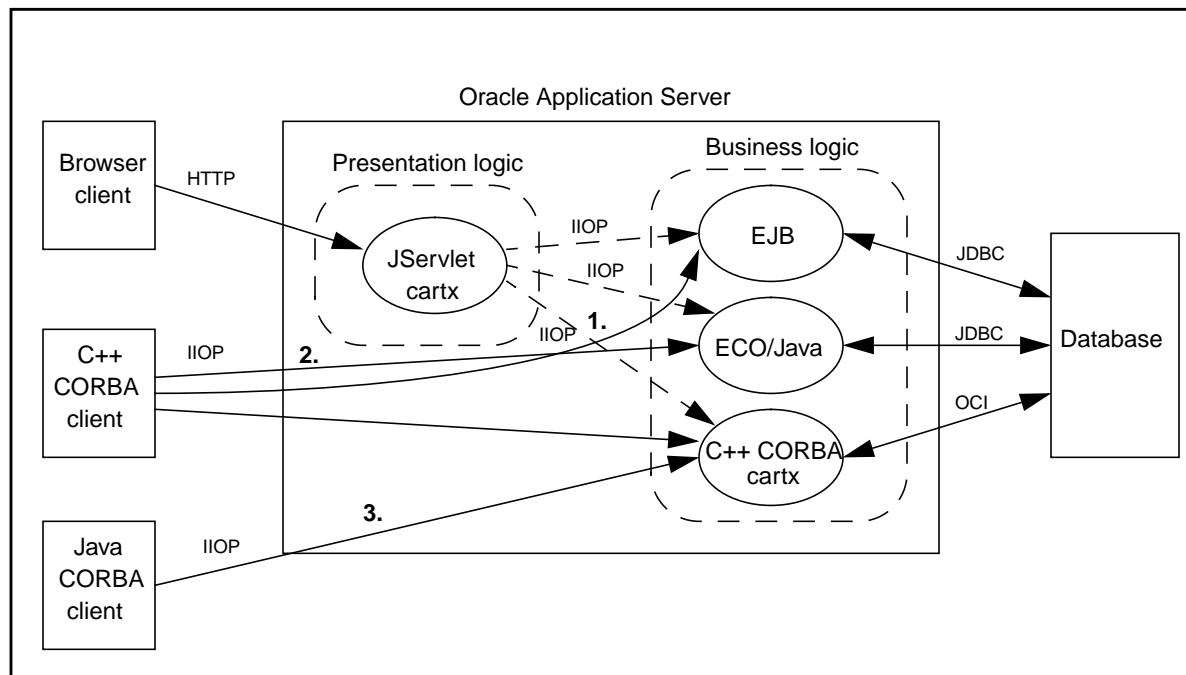
Table 1–1 compares the steps involved in the development of these two types of applications.

**Table 1–1 General CORBA vs. C++ CORBA cartridge application development**

Step	CORBA Application Development	C++ CORBA Cartridge Application Development
1	Determine application's objects and define their IDL interfaces.	Determine application's objects and define their IDL interfaces.
2	Compile the IDL into C++ stubs and skeletons.	Compile the IDL into C++ stubs and skeletons.
3	Declare and implement C++ servant classes for your CORBA objects.	Declare and implement C++ implementation objects for your C++ objects.
4	Write a server main program.	Managed by container.
5	Create the server executable.	Create the server module using <b>cppgen</b> and <b>cppinstaller</b> .
6	Write, compile, and link the client code with the generated stubs.	Write, compile, and link the client code with the generated stubs.

## Using C++ Cartridge in an N-tier Computing Model

Oracle Application Server provides the framework for an N-tier computing model. In an N-tier computing model, HTTP or IIOP clients form the first tier. The middle tier consists of various components, which provide presentation logic or business logic. The last tier is the database. This model offers a variety of communication paths between the client and the database depending on business requirements. Some of these communication paths are highlighted in [Figure 1–2](#).

**Figure 1-2 N-tier computing model**

The numbers in [Figure 1-2](#) are explained as follows:

1. A JServlet cartridge can access a C++ CORBA cartridge using JNDI over Cos-Naming (SPI).
2. A C++ CORBA client can access an ECO/Java application using IIOP.
3. A Java CORBA client can access a C++ CORBA cartridge using IIOP.

This chapter provides a step-by-step guide on how to create a C++ application. This includes writing the cartridge IDL files, writing the implementation object for the cartridge, creating the deployment descriptor, and registering the application with Oracle Application Server.

The C++ application in this tutorial is called “Bank”, and it contains one C++ cartridge called “Account”.

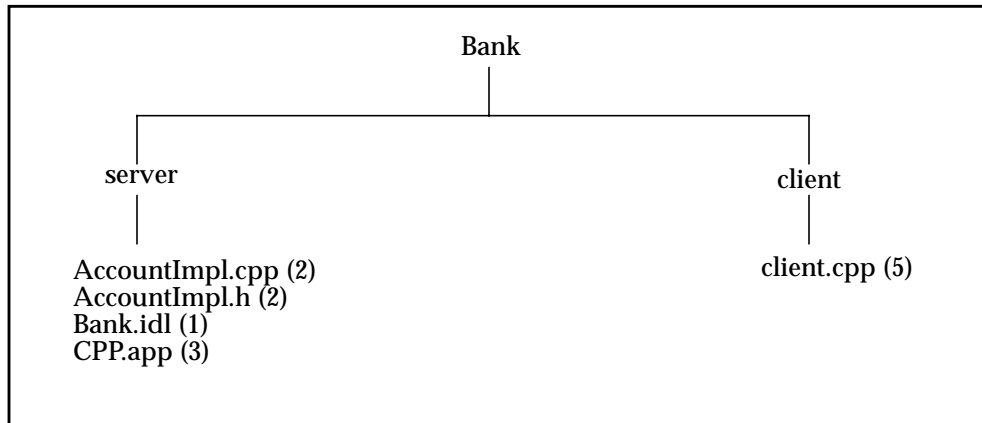
The tutorial covers the following steps:

1. Writing the Cartridge IDL File
2. Creating the Implementation Object
3. Creating the Deployment Descriptor File
4. Deploying the Application
5. Creating the Client Program
6. Creating the Client Executable
7. Running the Client to Access the C++ Application

## Files in the Tutorial

Create a directory structure as shown in Figure 2–1 to contain the files for the tutorial. The number beside each file indicates the step in which the file will be created.

**Figure 2–1** Directory structure for the tutorial



---

---

**Note:** This tutorial pertains only to deploying your application on the primary node of an Oracle Application Server site. For information on deploying your application on a secondary node, see Chapter 6, “Installing C++ Applications”

---

---

## 1. Writing the Cartridge IDL File

Create the following IDL file and save it as **Bank.idl** in the server directory.

```
//Bank Application
module Bank
{
    interface Account
    {
        short getBalance();
        void deposit(in short amount);
        void destroy();
    }
}
```

The IDL file contains the remote interfaces for all the cartridges in the C++ application. In **Bank.idl** the module Bank has an interface Account. The Account interface is implemented as a C++ cartridge.

---

**Note:** You can define the remote interfaces in multiple IDL files, but the application IDL file must include these individual IDL files.

---

## 2. Creating the Implementation Object

Save the following program as **AccountImpl.h** in the server directory:

```
#ifndef ACCOUNTIMPL_H
# define ACCOUNTIMPL_H

#ifdef CPP_ORACLE
# include <cpp.h>
#endif

/**
 * C++ Object Bank::Account implementation. The implementation is required
 * to extend the oas::cpp::Object base class
 */
class AccountImpl : public oas::cpp::Object
{
private:
    short                m_balance;
    oas::cpp::Context*   m_ctx;

public:
    AccountImpl() {}

    /* Implementation of the oas::cpp::Object methods. */
    void setContext(oas::cpp::Context* ctx);
    void cppCreate();
    void cppRemove();

    /* Implementation of the Bank::Account interface */
    short getBalance();
    void deposit(short amount);
    void destroy();
};

#endif
```

Create the following program and save it as **AccountImpl.cpp** in the server directory. This program is the C++ implementation object and must implement the `oas::cpp::Object` abstract base class provided by Oracle Application Server.

```
#ifndef ACCOUNT_H
# include <AccountImpl.h>
#endif

/**
 * The setContext method is invoked by the C++ cartridge container
 * with the object context for the specific cartridge instance
 *
 * @param IN oas::cpp::Context The context specific to the C++ object
 */
void AccountImpl::setContext(oas::cpp::Context* ctx)
{
    m_ctx = ctx;
}

/**
 * The instance specific initialization should be done in cppCreate.
 * If the instance initialization fails, the method is required to
 * log the failure and throw oas::cpp::CreateException.
 */
void AccountImpl::cppCreate()
{
    oas::cpp::Logger& log = m_ctx->getLogger();
    m_balance = 100;
    log << "Account instance created with initial balance of "
        << m_balance << "\n";
}

/**
 * The instance specific cleanup should be done here.
 */
void AccountImpl::cppRemove()
{
    oas::cpp::Logger& log = m_ctx->getLogger();

    m_balance = 0;
    log << "Account instance being destroyed\n";
}

/**
 * Bank::Account::getBalance implementation.
 */
```



```

    */
CORBA::Short AccountImpl::getBalance()
{
    oas::cpp::Logger& log = m_ctx->getLogger();

    log << "get Balance called returning "
        << m_balance << "\n";
    return m_balance;
}

/**
 * Bank::Account::deposit implementation
 */
void AccountImpl::deposit(CORBA::Short amount)
{
    oas::cpp::Logger& log = m_ctx->getLogger();

    log << "deposit called\n";
    m_balance = m_balance + amount;
    log << "new amount: " << m_balance << "\n";
}

/**
 * Bank::Account::destroy implementation
 */
void AccountImpl::destroy()
{
    oas::cpp::Logger& log = m_ctx->getLogger();

    log << "remove called\n";
    m_ctx->remove();
}

```

### 3. Creating the Deployment Descriptor File

The deployment descriptor file is a text file called **CPP.app**, and it contains information used by the Oracle Application Server Manager to register C++ applications to Oracle Application Server.

The file contains information such as the name of the application, the cartridges in the application, and the names of the remote interface classes.

Create the following **CPP.app** file in the server directory:

[APPLICATION]

```
name=Bank
transactions=Disabled

[Account]
stateless=false
remoteInterface=Bank::Account
implementationClass=AccountImpl
implementationHeader=AccountImpl.h
```

In one application, two or more cartridges cannot have the same name. See Chapter 4, “Creating the Deployment Descriptor File” for more on creating deployment descriptor files.

## 4. Deploying the Application

---

---

**Note:** Make sure that your environment variable `$ORACLE_HOME` and `$ORAWEB_HOME` are properly set. See *Oracle Application Server Installation Guide* for details.

---

---

To deploy your C++ cartridge application on an Oracle Application Server site, you will need to complete the following steps in the server directory:

1. Generate stubs and skeletons for your IDL file.

Use the following command to run the IDL C++ compiler (**oasoidlc**) to generate C++ stubs and skeletons for **Bank.idl**:

```
prompt>$ORACLE_HOME/orb/4.0/bin/oasoidlc -g cplus -I $ORACLE_HOME/orb/4.0/
public -I $ORAWEB_HOME/cpp/public -I . -A "oasoidlc.c-cplus-kwd=true"
Bank.idl
```

2. Generate the C++ cartridge factories.

You use the utility **cppgen** to create C++ cartridge factories. Use the following command to create the instance factories for each of the cartridges in the application:

```
prompt>$ORAWEB_HOME/bin/cppgen -o . -a CPP.app -i Bank.idl
```

### 3. Create and link the cartridge shared library.

Create the application shared library (**Bank.so** for Solaris) by linking in the object files **BankS.o**, **BankC.o**, **BankW.o**, **BankT.o**; and the cartridge implementation file **AccountImpl.o**. For example:

```
prompt>CC -c -Kpic -g -I$ORAWEB_HOME/cpp/public -I. -I$ORACLE_HOME/orb/4.0/  
public -I$ORACLE_HOME/ys/pub -o BankS.o BankS.cpp
```

```
prompt>CC -c -Kpic -g -I$ORAWEB_HOME/cpp/public -I. -I$ORACLE_HOME/orb/4.0/  
public -I$ORACLE_HOME/ys/pub -o BankC.o BankC.cpp
```

```
prompt>CC -c -Kpic -g -I$ORAWEB_HOME/cpp/public -I. -I$ORACLE_HOME/orb/4.0/  
public -I$ORACLE_HOME/ys/pub -o BankW.o BankW.cpp
```

```
prompt>CC -c -Kpic -g -I$ORAWEB_HOME/cpp/public -I. -I$ORACLE_HOME/orb/4.0/  
public -I$ORACLE_HOME/ys/pub -o BankT.o BankT.cpp
```

```
prompt>CC -c -Kpic -g -I$ORAWEB_HOME/cpp/public -I. -I$ORACLE_HOME/orb/4.0/  
public -I$ORACLE_HOME/ys/pub -o AccountImpl.o AccountImpl.cpp
```

```
prompt>CC -G -o Bank.so BankS.o BankC.o BankW.o BankT.o AccountImpl.o -  
L$ORAWEB_HOME/cpp/lib -lwrcc -lCstd -lCrun
```

### 4. Install the application on the Oracle Application Server site.

Use the following command to install the application shared library on the Oracle Application Server site:

```
prompt> $ORAWEB_HOME/bin/cppinstaller -f -a CPP.app -l Bank.so
```

This command registers your application with the Oracle Application Server Site Manager.

---

---

**Note:** For Solaris, you need C++ compiler version 5.0.

---

---

See Chapter 6, “Installing C++ Applications” for detailed explanations of the above mentioned steps.

## 5. Creating the Client Program

Create the following program and save it as **client.cpp** in the client directory. This program is the client code for accessing the C++ cartridge. The client uses CosNaming to access the C++ cartridge.

```
#ifndef CPP_ORACLE
#include <cpp.h>
#endif

#include <cosnamC.h>
#include <BankC.h>

int
main (int argc, char **argv)
{
    CORBA::ORB_var the_orb = CORBA::ORB_init(argc, argv);

    long i = 0;
    CosNaming::NamingContext_var rootNC_var = NULL;

    if (argc < 2)
    {
        cout << "Usage: " << argv[0] << " <oas://<host>:<port>" << endl;
        exit (1);
    }

    // Obtain the IOR for the root naming context using the
    // C++ cartridge Name Service boot strap mechanism. A stringified
    // object reference of the root naming context is returned
    // by the NSBootStrap object
    try
    {
        const char*          rootNCobj_ior = NULL;
        CORBA::Object_ptr    rootNCobj_ptr = NULL;

        CosNaming::NamingContext_ptr rootNC_ptr = NULL;

        rootNCobj_ior = oas::cpp::NSBootStrap::getOASRootNamingContext
            (argv[1]);
        rootNCobj_ptr = CORBA::ORB::string_to_object(rootNCobj_ior);
        oas::cpp::NSBootStrap::freeIOR (rootNCobj_ptr);
    }
```

```

    rootNC_ptr = CosNaming::NamingContext::_narrow(rootNCobj_ptr);
    if (CORBA::is_nil(rootNC_ptr))
    {
        cerr << "Error obtaining root naming context!" << endl;
        exit(1);
    }

    rootNC_var = rootNC_ptr;
}
catch (oas::cpp::URLFormatException &e)
{
    cerr << "Invalid URL format: " << argv[1] << endl;
    cerr << "Not understood from: " << e.getMessage() << endl;
    exit(1);
}
catch (oas::cpp::ListenerException &e)
{
    cerr << "Listener Error: Check if listener is running" << endl;
    exit(1);
}
catch (oas::cpp::OutOfFileDescriptorsException &e)
{
    cerr << "No more file descriptors" << endl;
    exit(1);
}
catch (oas::cpp::UnknownAddressException &e)
{
    cerr << "Address not known" << endl;
    exit(1);
}
catch (oas::cpp::ConnectionFailedException &e)
{
    cerr << "Connection failed. Try later" << endl;
    exit(1);
}
catch (oas::cpp::IOException &e)
{
    cerr << "IO Exception" << endl;
    exit(1);
}
catch (oas::cpp::InternalErrorException &e)
{
    cerr << "Internal error received: " << e.getMessage() << endl;
    exit(1);
}
}

```

```
catch (oas::cpp::Exception &e)
{
    cerr << "Unknown error received: " << e.getMessage() << endl;
    exit(1);
}
catch (...)
{
    cerr << "Unknown error obtaining root naming context" << endl;
    exit(1);
}

// Create a CosNaming name object to access the c++ cartridge instance
CosNaming::Name name;
name.length(2);
name[0].id = CORBA::string_dup("Bank");
name[1].id = CORBA::string_dup("Account");

try
{
    CORBA::Object_ptr cppobj_ptr = rootNC_var->resolve(name);
    if (CORBA::is_nil(cppobj_ptr))
    {
        cerr << "Error resolving Bank/Account" << endl;
        exit(1);
    }

    cout << "c++ object : " << cppobj_ptr << endl;
    Bank::Account_ptr account_ptr = Bank::Account::_narrow(cppobj_ptr);
    if (CORBA::is_nil(account_ptr))
    {
        cerr << "Error narrowing Account instance" << endl;
        exit(1);
    }

    //Get the smart var ptr for automatic release
    Bank::Account_var account_var = account_ptr;
    cout << "Balance: " << account_var->getBalance() << endl;
    account_var->deposit(100);
    cout << "Deposting: 100 bucks... " << endl;
    cout << "New balance: " << account_var->getBalance() << endl;
    account_var->destroy();
}
```

```

        catch (CORBA::SystemException& se)
        {
            cerr << "Error: " << se._repository_id() << endl;
            return 1;
        }
        catch (...)
        {
            cerr << "Unknown error" << endl;
            return 1;
        }

        return 0;
    }

```

## 6. Creating the Client Executable

To create your client executable, you will need to complete the following steps in the client directory:

1. Generate stubs of the standard CosNaming interface.

Use the following command to run the IDL C++ compiler (**oasoidlc**):

```

prompt>$ORACLE_HOME/orb/4.0/bin/oasoidlc -g cplus -A "oasoidlc.c-cplus-
kwd=true" $ORACLE_HOME/orb/4.0/public/cosnam.idl

```

2. Compile and link the client program with the required libraries and runtime files.

For Solaris:

```

$CC -c -g -I$ORAWEB_HOME/cpp/public -I. -I$ORACLE_HOME/orb/4.0/public -
I$ORACLE_HOME/ys/pub -I../server -o client.o client.cpp

```

```

$CC -c -g -I$ORAWEB_HOME/cpp/public -I. -I$ORACLE_HOME/orb/4.0/public -
I$ORACLE_HOME/ys/pub -I../server -o cosnamC.o cosnamC.cpp

```

```

$CC -o client client.o ../server/BankC.o cosnamC.o $ORACLE_HOME/orb/4.0/lib/
liborb.so $ORACLE_HOME/orb/4.0/lib/libyop.so $ORAWEB_HOME/cpp/lib/
libcppcl.so -lthread -lsocket

```

See Chapter 6, “Installing C++ Applications” for detailed explanations of the above mentioned steps.

## 7. Running the Client to Access the C++ Application

Reload the Oracle Application Server site as follows:

```
prompt> owsctl start
```

(If Oracle Application Server is not running.)

or,

```
prompt> owsctl reload -w all
```

(If Oracle Application Server is already running.)

Note that every time you register a new application with an Oracle Application Server site, you should reload the application server using the above command.

Run the client program by giving the following command:

```
prompt> client oas://host-name:port
```

where,

- *host-name* is the name of the machine on which Oracle Application Server is running.
- *port* is the port at which the Web listener is listening.

The client does a CosNaming `resolve()` and acquires the object reference of the C++ implementation object. The Account methods are now invoked on the instance. Finally, the client invokes the `destroy()` method to destroy the cartridge instance.

Here is a sample run:

```
prompt> client oas://isp-sun13.us.oracle.com:80
```

```
Balance: 100
```

```
Deposting: 100 bucks...
```

```
New balance: 200
```



---

# Developing C++ Cartridges

To develop a C++ cartridge, you need to create the cartridge remote interface and the C++ implementation object. This chapter describes how to create these two entities.

## Contents

- [Cartridge Remote Interface](#)
- [C++ Implementation Object](#)
- [Logging](#)
- [Transactions](#)
- [Cartridge Environment](#)
- [Stateful and Stateless Cartridges](#)

## Cartridge Remote Interface

Once you complete the design phase of your application and identify all objects, you need to map all the objects to C++ cartridges. This section gives some guidelines for mapping the objects to C++ cartridges.

You create a cartridge remote interface for your application using interface definition language (IDL) as specified in CORBA specification 2.2. This IDL file defines the cartridge remote interface of your application. A module in this IDL file corresponds to your C++ application. Each interface in a module maps to a C++ cartridge. A client acquires the object reference of a C++ object by specifying the application and cartridge names.

Note that you need to have a single IDL file for one application.

Following is an example of an IDL file that defines the remote interface for the cartridge Employee in an application called HR:

```
module HR
{
    exception DBError
    {
        string ermesg;
    };

    struct EmployeeRecord
    {
        short    id;
        string    name;
        string    job;
    };

    interface Employee
    {
        EmployeeRecord getEmployeeRecord(in short id)
            raises (DBError);

        void updateEmployeeRecord(in short id, in EmployeeRecord rec)
            raises (DBError);

        void destroy();
    };
};
```

The remote interface for the cartridge Employee has the following C++ methods:

- `getEmployeeRecord()`
- `updateEmployeeRecord()`
- `destroy()`

The implementation of these methods is provided by the C++ implementation objects.

## C++ Implementation Object

For each interface defined in the cartridge remote interface, you should provide a C++ implementation object. The C++ implementation object implements the business logic of the C++ cartridge. In other words, the C++ implementation object is

the C++ code that you write on the server side. The remote calls from the client are delegated to this object.

Oracle Application Server provides [oas::cpp::Object Base Class](#) and [oas::cpp::Context Object](#) as the basic classes for developing C++ applications. See [Chapter 7, “Reference”](#) for a description of all available classes.

## oas::cpp::Object Base Class

Each C++ implementation object should inherit the base class `oas::cpp::Object` and implement the following methods:

- `setContext()`: The container invokes this method on the C++ implementation object and passes along the instance specific context to this object.
- `cppCreate()`: The container invokes this method on the implementation object after the `setContext()` is called. Implementation object specific initialization should be done here. The logger object is available for logging at this stage. If an error occurs in initializing, the implementation object can throw `oas::cpp::CreateException` exception. When this exception is thrown, the container destroys the implementation object.
- `cppRemove()`: The container invokes this method on the C++ object before it **deletes** the implementation object. Instance specific garbage collection should be done here. The implementation object can be removed for two reasons:
  - The implementation object has timed out. See [Chapter 4, “Creating the Deployment Descriptor File”](#) for more on the timeout parameter associated with a C++ cartridge.
  - The `remove()` method was invoked on the `oas::cpp::Context` object.

The following code snippet shows how to use the above mentioned methods:

```
class EmployeeImpl: public oas::cpp::Object
{
public:
    EmployeeImpl() {};
    virtual ~EmployeeImpl() {};

    virtual HR::EmployeeRecord* getEmployeeRecord(int id);
    virtual void updateEmployeeRecord(int id, HR::EmployeeRecord_out rec);

    void setContext(oas::cpp::Context* ctx)
    {
        _ctx = ctx;
```

```
    }

    void cppCreate();
    void cppRemove();
    void destroy()
    {
        if (_ctx != NULL)
            _ctx->remove();
    }
private:
    oas::cpp::Context* _ctx;
    boolean            _success;
    DbUtil*            _dbUtil;
};
```

The C++ implementation object should also implement all the business logic methods. The signatures for these methods should match with those of the C++ mapping of the remote interface (the IDL file) of the cartridge.

## **oas::cpp::Context Object**

The container creates a Context object for every C++ implementation object, which provides the implementation object with access to the container services, such as logging, transactions, and cartridge environment. The methods on the Context object are defined as follows:

- `getLogger()` : Returns a reference to the logger for this cartridge instance.
- `getUserTransaction()` : Returns a user transaction object.
- `getEnvironment()` : Returns the cartridge environment name value pairs.
- `getObject()` : Returns the object reference for the C++ object.
- `remove()` : Destroys the C++ implementation object.

## **Logging**

The container creates a new logger instance for every C++ implementation object. A reference to the logger can be obtained from the object context.

## **Overview**

Each message can be logged at different severity levels. Following is the list of different severity levels:

- SEVERITY\_FATAL: Use this severity level if a fatal error (that will result in destroying the C++ cartridge) occurs. An example of a fatal error is failure of the initialization of the C++ object.
- SEVERITY\_WARNING: Use this severity level for reporting warnings.
- SEVERITY\_INITTERM: Use this severity level for reporting initialization and termination of C++ objects.
- SEVERITY\_DEFAULT: This severity level is automatically set if you do not set any severity level.
- SEVERITY\_TRACE: Use this severity level for debugging your C++ cartridge.

The message severity levels are numbers in the range of 0 to 15. You can set the Oracle Application Server logger daemon to a specific severity level from the Oracle Application Server Manager. All the messages whose severity is lesser in value than that set on the logger daemon are logged. By default, error messages are logged to the log file. You can change the location of the log file for your application using the Oracle Application Server Manager. See *Oracle Application Server Administration Guide* for details.

The log messages are flushed to the logger daemon when a new line character `\n` appears in the message buffer. The messages are also flushed when the instance is destroyed.

## Example

The following code snippet gives an example of how to use logging:

```
void EmployeeImpl::cppCreate()
{
    oas::cpp::Logger& log = _ctx->getLogger();

    _success = TRUE;
    try
    {
        log << "Creating a new DBUtil\n"; //DBUtil is a utility class for
        //creating a database connection
        _dbUtil = new DbUtil("scott", "tiger", "test"); //creating OCI
        // connection
        log << "new DB util created\n";
    }
}
```

```
        catch (const char* e)
        {
            cerr << e << endl;
            log << "DBUtil Error: " << e << "\n";
            _success = FALSE;
        }
    }

void EmployeeImpl::cppRemove()
{
    delete _dbUtil;
}

HR::EmployeeRecord* EmployeeImpl::getEmployeeRecord(int id)
{
    oas::cpp::Logger& log = _ctx->getLogger();
    Employee* e = NULL;
    if (_success == FALSE)
        return NULL;

    try
    {
        log << "Getting info for employee " << id << "\n";
        e = _dbUtil->getEmployee(id);
        log << "Got info successfully\n";
    }
    catch (const char* err)
    {
        log << "Error! " << err << "\n"; //logging error messages
        return NULL;
    }

    if (e != NULL)
    {
        HR::EmployeeRecord* er = new HR::EmployeeRecord;
        er->id = e->getId();
        er->name = CORBA::string_dup(e->getName());
        er->job = CORBA::string_dup(e->getJob());
        return er;
    }

    return NULL;
}

void EmployeeImpl::updateEmployeeRecord(int id, HR::EmployeeRecord_out er)
```

## Transactions

The C++ cartridge supports distributed transactions. You can write an application that updates data in multiple databases, distributed across multiple cartridges in a single global transaction.

### Overview

The C++ cartridge transactions are modeled after OMG's Object Transaction Service (OTS) 1.1 specification. For enabling transactions in your application, you will need to configure transactional database access descriptors (DADs) and the distributed transaction coordinator (DTC). For more information, see the *Oracle Application Server Administration Guide*.

You can do transaction demarcation through the `UserTransaction` object that is obtained from the `Context` object. The `UserTransaction` object supports the following methods:

- `begin()`: Begins a global transaction.
- `commit()`: Commits the global transaction. You should do a commit in the same cartridge that began the transaction.
- `rollback()`: Rollsback the existing active transaction.
- `getStatus()`: Gets the transaction status of the C++ implementation object.
- `setTransactionTimeout()`: Modifies the value of timeout that is associated with the transaction started by the current cartridge with the `begin` method.
- `setRollbackOnly()`: Modifies the current transaction such that the only possible outcome of the transaction is to rollback.

See [Chapter 7, "Reference"](#) for a complete description of these methods.

---

---

**Note:** Client-side transaction demarcation is not supported in the C++ cartridge.

---

---

### Example

The following code snippet is from the IDL file of an application that uses distributed transactions:

```
interface Employee: CosTransaction::TransactionObject
{
    EmployeeRecord getEmployeeRecord(in short id, in string url)
```

```
        raises (DBError);

void updateEmployeeRecord(in EmployeeRecord rec)
    raises (DBError);

void destroy();
};
```

The following code snippet is an example of a C++ implementation object that uses transactions:

```
EmployeeRecord* EmployeeImpl::getEmployeeRecord(int id)
    throw (HR::DBError)
{
    Employee* e = NULL;
    try
    {
        oas::cpp::transaction::UserTransaction* ut = _ctx->getUserTransaction();
        if (ut == NULL)
            throw "Panic: userTransaction is null";

        ut->begin();
        _dbUtil = new DbUtil("JCOTEST");

        cout << "Begin transaction successful" << endl;
        e = _dbUtil->updateEmployee(id);
        ut->commit();

        cout << "Commit transaction successful" << endl;
        delete _dbUtil;
        _dbUtil = NULL;
    }
    catch (const char* err)
    {
        :
    }
```

## Cartridge Environment

You can associate name value pairs (environment parameters) with the cartridge in the deployment descriptor file. See [Chapter 4, “Creating the Deployment Descriptor File”](#). After you deploy the application, these name value pairs can be customized through the Oracle Application Server Manager.



## Overview

The `oas::cpp::Environment` class provides a C++ implementation object with access to the C++ cartridge's environment parameters. The cartridge can either look for a particular parameter name and get its value using the `getParameterByName()` method, or get all the configured parameters using an index via the `getParameterByIndex()` and `getNameByIndex()` methods. See [Chapter 7, "Reference"](#) for details.

## Example

```
void EnvironmentImpl::cppCreate()
{
    oas::cpp::Logger& log = _ctx->getLogger();
    log << "In cppCreate\n";

    _env = _ctx->getEnvironment();
    _numEnv = _env->length();

    log << "Num Env: " << _numEnv << "\n";
}

char* EnvironmentImpl::getEnvironmentAt(CORBA::Short index)
{
    oas::cpp::Logger& log = _ctx->getLogger();

    if (index >= _numEnv)
        return NULL;

    const char *name = _env->getNameByIndex(index);
    const char *value = _env->getParameterByIndex(index);

    int length = strlen (name) +
        strlen (value) +
        1; // the plus 1 is for the '=' character
    char *retval = CORBA::string_alloc (length);
    char *retvall = retval;

    while (*name)
    {
        *retvall++ = *name++;
    }
    *retvall++ = '=';
}
```

```

while (*value)
{
    *retvall++ = *value++;
}
*retvall++ = 0;

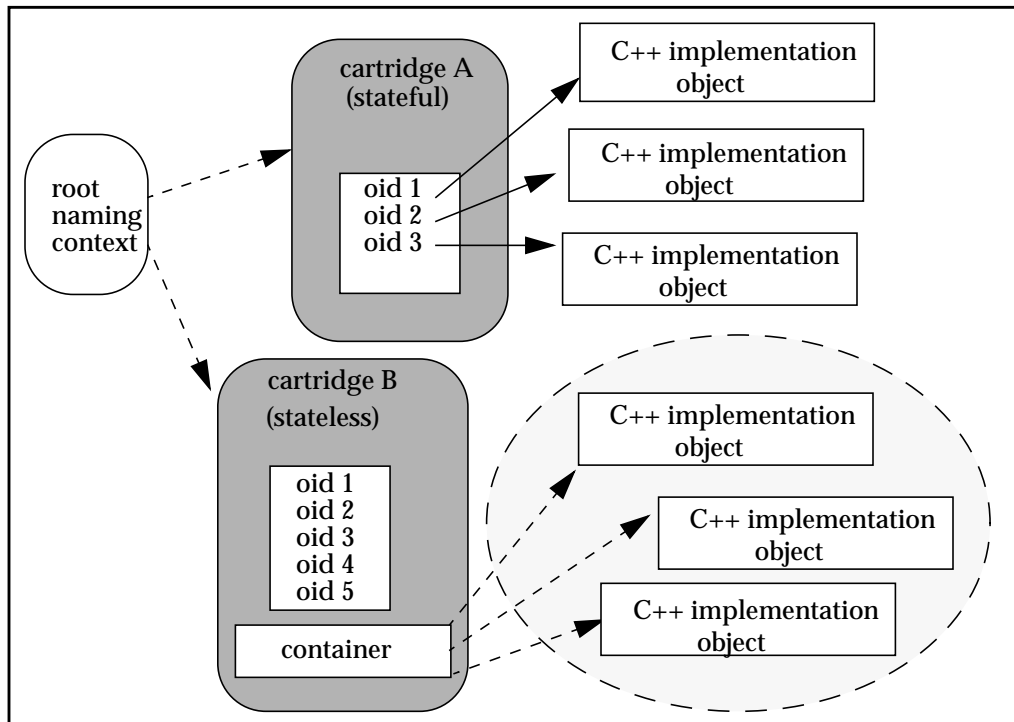
return retval;
}
:

```

## Stateful and Stateless Cartridges

You can specify C++ cartridges as [Stateful Cartridges](#) or [Stateless Cartridges](#) in the C++ application deployment descriptor file. By default a C++ cartridge is stateful. Figure 3–1 describes the container architecture.

**Figure 3–1** *Stateful and stateless cartridges*

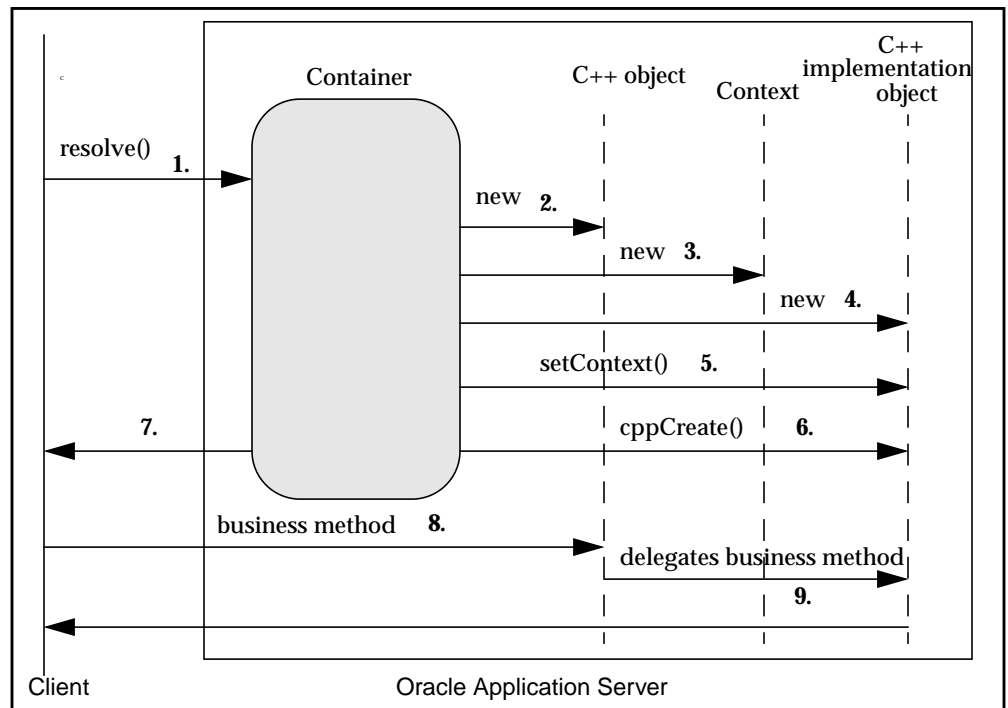


## Stateful Cartridges

In a stateful cartridge one C++ implementation object (which is an instance of the C++ implementation class that you write) is created every time a client invokes a `resolve()` method on the leaf node of the naming tree. (See [Chapter 1, “Overview”](#) for more on the naming tree.) The C++ implementation object is tied to the client until the `remove()` method is invoked on the Context object or the implementation object’s timeout.

In a stateful cartridge the C++ implementation object is bound to an object id (oid). See Figure 3–1. Stateful cartridges allow a client to create a conversational state associated with a C++ implementation object, which is retained as long as the implementation object is alive. However, this comes with an associated cost since a new C++ implementation object is created for each client. Figure 3–2 shows the sequence diagram for stateful C++ cartridges.

**Figure 3–2** Sequence diagram for stateful C++ cartridges



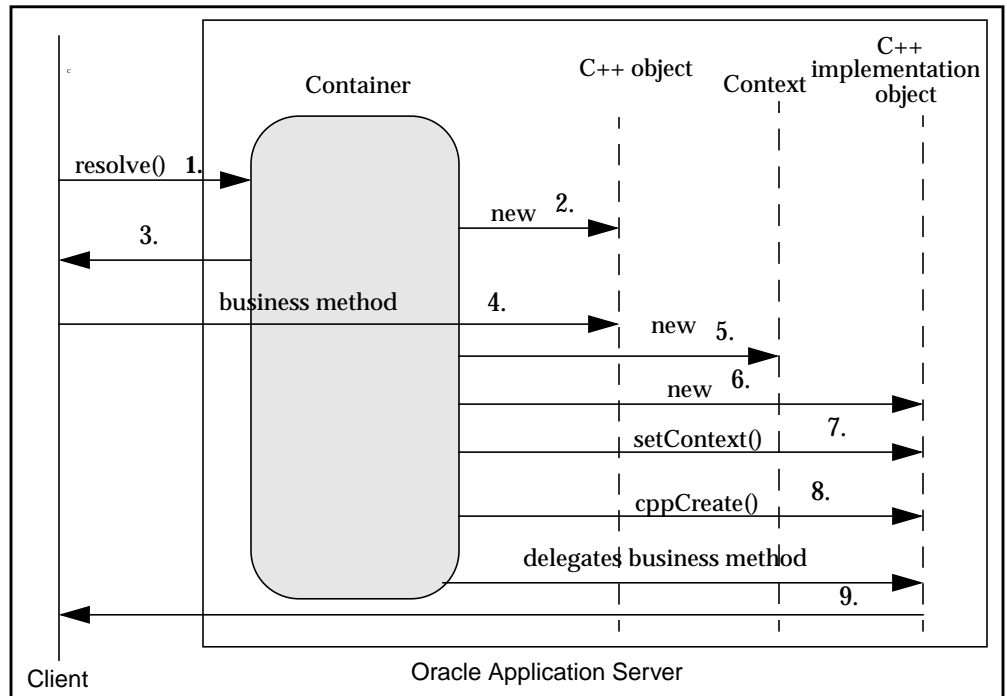
The numbers in Figure 3–2 are explained as follows:

1. The client does a `CosNaming::resolve()` to obtain the object reference to a C++ object.
2. The container creates an instance of the C++ object.
3. A new Context object is created.
4. A C++ implementation object is instantiated for this C++ object.
5. The container invokes `setContext()` on the implementation object.
6. The container invokes `cppCreate()`.
7. The object reference for the C++ object is returned to the client.
8. The client invokes a business method on the C++ object.
9. The C++ object delegates the business method to the implementation object, which returns the result to the client.

## Stateless Cartridges

A C++ cartridge can also be declared stateless. Thus, a client cannot associate conversational state with the C++ implementation object. The container creates a pool of stateless implementation objects on demand. When the client invokes `resolve()` to obtain a stateless cartridge, only a reference is created to the C++ cartridge. When a method is invoked on this reference, the container assigns an implementation object from the pool (Figure 3–1), or creates a new instance of the implementation object.

Stateless cartridges allow for building scalable servers by letting a large number of clients to be serviced by a limited set of C++ implementation objects. Figure 3–3 shows the sequence diagram for stateless C++ cartridges.

**Figure 3–3** Sequence diagram for stateless C++ cartridges

The numbers in Figure 3–3 are explained as follows:

1. The client does a `CosNaming resolve()` to obtain the object reference to a C++ object.
2. The container creates an instance of the C++ object.
3. The object reference for the C++ object is returned to the client.
4. The client invokes a business method on the C++ object.
5. A new Context object is created.
6. A C++ implementation object is instantiated for this C++ object.
7. The container invokes `setContext()` on the implementation object.
8. The container invokes `cppCreate()`.
9. The container delegates the business method to the implementation object, which returns the result to the client.



---

# Creating the Deployment Descriptor File

This chapter describes how to create the deployment descriptor file for C++ applications. By convention, the deployment descriptor file is named **CPP.app**. You create one file per application, which is a collection of one or more C++ cartridge components.

## Contents

- [Overview](#)
- [Structure of the Deployment Descriptor File](#)

## Overview

Once the C++ application is developed, you need to create a deployment descriptor file that describes the properties of the C++ cartridges. The deployment descriptor file provides information about the application, such as the name of the application, the cartridges in it, and other information related to the cartridges.

Some of the values in the file are used as the default values for the application. You can change these values after you have installed the application in the application server.

## Structure of the Deployment Descriptor File

The deployment descriptor file contains the following sections:

- [Application Section](#)
- [Cartridge Section](#)

Each cartridge in the application has its own [`<cartridgeName>`] section. For example, if you have three cartridges in a C++ application, you would have one [`APPLICATION`] section, and three [`<cartridgeName>`] sections.

Each section contains property name-value pairs of the form:

```
<propname> = <value>
```

*propname* cannot contain space characters. Note that the contents of the file are case-sensitive.

Lines starting with a semicolon character (“;”) are comments, and the whole line is ignored.

Note that this file is called **CPP.app** by convention: it can have any other name.

## Application Section

The application section contains entries for the entire application. Following is an example of the application section of **CPP.app** for the bank application:

```
[APPLICATION]
name = bank
timeout = 6000
transactions = Enabled
transactionalDads = BANK
authenticationString = Basic
```

Table 4–1 describes the properties of the [`APPLICATION`] section:

**Table 4–1    Application properties**

Property	Description
name	The name of the application. The name cannot contain “.” or white space characters. This name is used by clients to identify the application. (Clients identify the application using CosNaming or JNDI.)  This property is required.



**Table 4–1 Application properties**

Property	Description
timeout	<p>Application level timeout. How long in seconds an object can be idle before it is destroyed.</p> <p>When a C++ object has been idle for the specified duration (for example, the client has not made a request for this specified amount of time), the container can sever the connection between the client and the C++ object. After severing the connection, container can use the C++ object to service another client or it can destroy the C++ object.</p> <p>If the timeout is not set or if it is set to 0, then no timeout processing is performed. This timeout feature is intended to free up instances when clients terminate abnormally and do not get to release the C++ object.</p> <p>This property is optional. Default is 24 hours.</p>
transactions	<p>The possible values are Enabled/Disabled. If transactions are enabled, OTS is initialized when the server process comes up.</p> <p>This property is optional. Default is Disabled.</p>
transactionalDads	<p>This property is required if transactions are enabled. The value for this property is a comma-separated list of transactional databases used by the application.</p>
authenticationString	<p>The string that describes the authentication realm with which the C++ application is secured.</p> <p>This property is optional. By default, an application does not require any authentication.</p>

## Cartridge Section

Each cartridge in the application must have its own section in the **CPP.app** file. Following is example of the cartridge section of **CPP.app** for the bank application:

```
[Account]
remoteInterface = Bank::Account
implementationClass = Bank::AccountImpl
implementationHeader = AccountImpl.h
timeout = 1000
stateless = false
authenticationString = Basic(BasicRealm)
```

```
[Account.ENV]
name1 = value1
name2 = value2
```

The name of the cartridge is specified between the square brackets. Clients use this name to identify the C++ cartridge that they want to access. The cartridge name cannot contain the “.” character.

Table 4–2 describes the properties in the [`<cartridgeName>`] section.

**Table 4–2    Cartridge properties**

Property	Description
remoteInterface	<p>The name of the cartridge remote interface. The deployment tools described in <a href="#">Chapter 6, “Installing C++ Applications”</a> look for this interface in the IDL file.</p> <p>This property is required.</p>
implementation-Class	<p>The servant implementation of the remote interface. Inheritance-based implementation is used from implementing the remote interface, and therefore, this class is required to inherit from the skeleton classes generated by the IDL compiler.</p> <p>This property is required.</p>
implementation-Header	<p>The name of the header file where the implementation class is defined.</p> <p>This property is required.</p>
timeout	<p>Cartridge specific timeout value.</p> <p>This property is optional. If no timeout value is specified, it defaults to the application timeout.</p>
stateless	<p>Whether the cartridge is stateful or stateless. The possible values for this property are True/False.</p> <p>This property is optional. Default is False.</p>
authenticationString	<p>The string that describes the authentication realm with which the C++ object is secured.</p> <p>This property is optional. If the authentication string is not specified, this property will take its value from the application authentication string. If cartridge authentication string is specified, it will take precedence over the application authentication string.</p>

There are no pre-defined system properties for the [`<cartridgeName.ENV`] section. This section is optional and contains cartridge-specific name value pairs. For example, if you are creating a mortgage application, you could have a property that specifies the interest rate:

```
[Mortgage.ENV]  
interestRate = 8.0
```

The `oas::cpp::Environment` class provides a C++ implementation object with access to the C++ cartridge's name value pairs. See [Chapter 3, “Developing C++ Cartridges”](#) for details.



---

# Developing Clients for C++ Applications

This chapter describes how to develop clients for C++ applications. See [Chapter 3, “Developing C++ Cartridges”](#) and [Chapter 4, “Creating the Deployment Descriptor File”](#) for information on developing and deploying C++ applications in the Oracle Application Server environment.

## Contents

- [Overview](#)
- [Client Side Object Request Broker \(ORB\)](#)
- [Getting the Object Reference for a C++ Object](#)
- [Using the C++ Cartridge](#)
- [Security](#)
- [Example](#)

## Overview

Developing clients for C++ application is very similar to creating clients for CORBA objects in general. You can use any client side ORB (compliant with CORBA 2.0 specification) for developing clients for C++ applications. Clients of C++ applications can be any of the following:

- C++ objects in the same or other C++ applications
- Java applets running in browsers
- Java applications
- JServlet cartridges

C++ and Java clients can access the C++ cartridge as follows:

- **C++ Clients** use the CORBA naming service to obtain C++ object references. The mechanism of obtaining the root naming context of the CORBA naming service running in Oracle Application Server is Oracle proprietary.
- **Java Clients** use JNDI to obtain the reference of a C++ object. The JNDI SPI implementation is provided with Oracle Application Server.

## Client Side Object Request Broker (ORB)

To access C++ objects, you will need an ORB on the client side. You can use any ORB that is CORBA 2.0 compliant. If you are using the ORB in Oracle Application Server, see “Object Request Broker Administration” in the *Oracle Application Server Administration Guide* on how to set up and configure the ORACLE ORB.

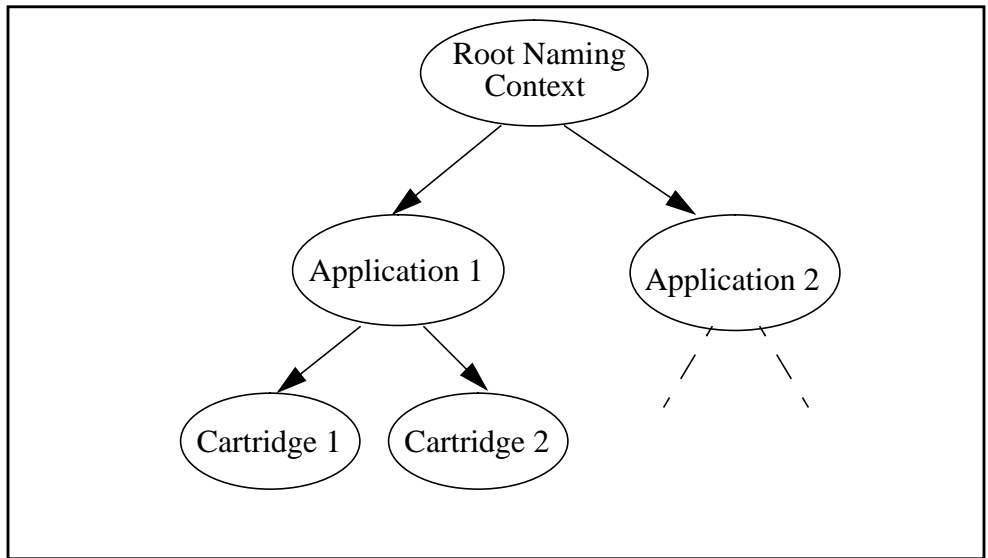
## Getting the Object Reference for a C++ Object

Before a client can invoke a method on a C++ object, it must get a reference to that object. Clients obtain a reference to the C++ object via the Oracle Application Server naming tree.

### The Naming Tree

The naming tree consists of the names of all the C++ applications and cartridges deployed in the Oracle Application Server. C++ applications and cartridges are represented by nodes in the tree. The node corresponding to a C++ application is the parent of the nodes corresponding to all the C++ cartridges contained in that application.

[Figure 5–1](#) is a diagrammatic representation of the naming tree.

**Figure 5–1** A simple naming tree

A C++ application name can be split into sub-components. For example, a C++ application name can have two sub-components: app1 and app2. The name of this application will be "app1/app2". The naming tree will show "app2" as a sub-node of "app1". Similarly, cartridge names can also be split.

The naming tree is a **transient** naming tree, that is, it is not persistent. It is created every time you start Oracle Application Server.

The naming tree is **read-only**. Clients can only read from the tree, they cannot bind or add to the naming tree. If clients try to perform any operation that can change the naming tree, they will get an exception like NO\_PERMISSION.

Clients written in C++ and Java use this naming tree. C++ clients use CosNaming to access the naming tree. Java clients use JNDI to access the naming tree. When clients perform a `resolve()` or a `lookup()` on a C++ cartridge name, they get the object reference of a new C++ object. The client can make method invocations on the C++ object. When the client does not need the object anymore, it should destroy the C++ object.

## Bootstrapping

This section explains how clients get to the root of the naming tree. Once clients get to the root, they can browse the naming tree by using the standard methods in CosNaming or JNDI. In the CORBA world, this boot strapping is generally achieved by the `resolve_initial_references()` method. Since this method is not interoperable with other ORBs, Oracle provides a proprietary mechanism for bootstrapping. This mechanism is independent of the ORB. It requires an Oracle Application Server listener. This mechanism is different for Java and C++ clients:

- [C++ Clients](#)
- [Java Clients](#)

### C++ Clients

C++ clients use CosNaming to access the naming tree. The clients can be either of the following:

- Remote clients: These run outside the Oracle Application Server site. They are stand-alone C++ applications, or C++ cartridges running on another Oracle Application Server site.
- Local clients: These run in the same Oracle Application Server site. They are stand-alone C++ applications, or another C++ cartridge. Local clients do not need a listener to access the naming tree.

**Included Files** You include the header file **wrccobs.h** in your client code. While linking the client, add the **libcppcl.so/libcppcl.dll** in the link line. These files are located in **\$ORAWEB\_HOME/cpp/lib/libcppcl.so** and **\$ORAWEB\_HOME/cpp/public/wrccobs.h**.

A copy of the OMG standard CosNaming interface is available in **\$ORACLE\_HOME/orb/4.0/public/cosnam.idl**. Generate the stubs for **cosnam.idl** using the IDL compiler of the client side ORB, and include them in the client program.

**Accessing the Naming Tree** You use the **OASNSBootStrap** class for both remote and local clients to access the root of the naming tree. This class consists of the following static methods:

- `const char *getOASRootNamingContext (const char *url)`

Used by remote clients. The url is of the form “**oas://<host>:<port>**”, where *host* is the machine on which the Oracle Application Server listener is running,



and *port* is the port on which the listener is listening. This method returns a stringified IOR of the root of the naming tree.

- `const char *getOASRootNamingContext ( )`  
Used by local clients. The return value is a stringified IOR of the root of the naming tree.
- `void freeIOR (const char *ior)`  
Used to free the memory occupied by the stringified IOR returned by the `getOASRootNamingContext ( )` methods. A memory leak will occur, if the client code does not call `freeIOR()`.

You use the stringified IOR returned by the `getOASRootNamingContext ( )` methods as the IOR of the `CosNaming::NamingContext` interface. Call `string_to_object ( )` on the IOR to get a `CORBA::Object` and then narrow the `CORBA::Object` to a `CosNaming::NamingContext` object.

**Exceptions** The `getOASRootNamingContext ( )` methods can throw many exceptions. Situations which cause an exception to be thrown include:

- Remote client fails to give the URL in the correct format.
- Remote client uses the `getOASRootNamingContext ( )` method without the URL.

All exceptions thrown by `getOASRootNamingContext ( )` derive from a base class called `OASException`. An exception may or may not have a message associated with it. If an exception has a message, you can retrieve it using the `getMessage ( )` method in the `OASException` class. See [Chapter 7, “Reference”](#) for more on the exceptions thrown by the `getOASRootNamingContext ( )` methods.

Note that a local client can access the root of the naming tree via the Oracle Application Server listener like a remote client

## Java Clients

Java clients use JNDI to access the naming tree. The bootstrapping mechanism (accessing the root of the naming tree) is a JNDI compliant mechanism. The clients can be either of the following:

- Remote clients: These need an Oracle Application Server listener. An example of a remote client is a java applet.
- Local clients: These do not need a listener. Example of local clients are EJB beans, ECO beans, and JServlet cartridges.

Java clients access C++ applications similar to how they access ECO applications. The only difference is that there is no concept of a home interface in C++ applications. In ECO applications, Java clients call `JNDI lookup()` on a ECO home name to get the object reference of a ECO home object. In C++ applications, Java clients call `JNDI lookup()` on a C++ cartridge name to get the object reference of the C++ object. Every time the client does a `lookup()`, it gets a new C++ object.

Java clients should have the file `$ORAWEB_HOME/classes/ecoapi.jar` in their CLASSPATH to access C++ cartridges in the naming tree.

See *Oracle Application Server Developer's Guide: EJB, ECO/Java and CORBA Applications* for details. Refer to the Javasoft website [www.javasoft.com](http://www.javasoft.com) for the JNDI API.

## Using the C++ Cartridge

You should include the stubs of the cartridge IDL file in the client program. See [Chapter 6, “Installing C++ Applications”](#) for how to use the Oracle IDL C++ compiler (`oasoidlc`).

### Invoking Methods on the C++ object

CosNaming or JNDI generates an object reference for `CORBA::Object` object. You narrow it to the required C++ object using the helper functions in the generated stubs of the C++ cartridge.

Now the client can directly invoke methods on the C++ object. Only one client is allowed to access a C++ object at a time. If the client is multi-threaded, and two clients try to access a C++ object simultaneously, one of them will get a `CORBA::TRANSIENT()` exception. If a client receives this exception, it should reissue the method call.

### Destroying the C++ object

The client cannot directly destroy the C++ object. The C++ implementation object destroys it by calling the `remove()` method on the Context object, which is given to each implementation object. The client makes a method call on the C++ object, and the method implementation calls `remove()` on the Context object to complete the destruction of the C++ object.

When an implementation object calls `remove()` on its Context object in one of its method invocations, it only destroys the C++ object on which the method invocation was made. In the case of stateful C++ cartridges, this will eventually destroy the C++ implementation object also. But in the case of stateless C++ cartridges,

destroying a C++ object does not automatically destroy the C++ implementation object.

Object references are not reused between clients. Each client gets a brand new object reference every time it performs `resolve()` or `lookup()` on the C++ cartridge name.

If you do not destroy a C++ object, it will remain active until it times out. The timeout value of each C++ cartridge is configurable.

## Security

Oracle Application Server provides a comprehensive set of security features for protecting IIOP-based applications, including C++ CORBA applications. Once a C++ cartridge is configured for protection, the client has to supply its credentials to get the object reference of the corresponding C++ object. Java clients of a protected C++ cartridge can use the JNDI interface to supply these credentials.

C++ clients of a protected C++ cartridge cannot use the CORBA CosNaming interface, since the CORBA CosNaming interface does not provide explicit support for passing security parameters. Oracle Application Server has extended the CosNaming interface into a new interface called **SecNamingContext**.

See the “Security for IIOP-based Applications: EJB, ECO/Java, and C++” chapter in the *Oracle Application Server Security Guide* for more on the various authentication mechanisms.

## Example

This program is the client code for accessing the C++ cartridge of the Bank application. The client uses CosNaming to access the C++ cartridge.

```
#ifndef CPP_ORACLE
#include <cpp.h>
#endif

#include <cosnamC.h>
#include <BankC.h>

int
main (int argc, char **argv)
{
    CORBA::ORB_var the_orb = CORBA::ORB_init(argc, argv);

    long i = 0;
    CosNaming::NamingContext_var rootNC_var = NULL;
```

```

if (argc < 2)
{
    cout << "Usage: " << argv[0] << " <as://<host>:<port>" << endl;
    exit (1);
}

// Obtain the IOR for the root naming context using the
// C++ cartridge Name Service boot strap mechanism. A stringified
// object reference of the root naming context is returned
// by the NSBootStrap object
try
{
    const char*          rootNCobj_ior = NULL;
    CORBA::Object_ptr    rootNCobj_ptr = NULL;

    CosNaming::NamingContext_ptr rootNC_ptr = NULL;

    rootNCobj_ior = oas::cpp::NSBootStrap::getOASRootNamingContext
        (argv[1]);
    rootNCobj_ptr = CORBA::ORB::string_to_object(rootNCobj_ior);
    oas::cpp::NSBootStrap::freeIOR (rootNCobj_ior);

    rootNC_ptr = CosNaming::NamingContext::_narrow(rootNCobj_ptr);
    if (CORBA::is_nil(rootNC_ptr))
    {
        cerr << "Error obtaining root naming context!" << endl;
        exit(1);
    }

    rootNC_var = rootNC_ptr;
}
catch (oas::cpp::URLFormatException &e)
{
    cerr << "Invalid URL format: " << argv[1] << endl;
    cerr << "Not understood from: " << e.getMessage() << endl;
    exit(1);
}
catch (oas::cpp::ListenerException &e)
{
    cerr << "Listener Error: Check if listener is running" << endl;
    exit(1);
}

```

---

```

catch (oas::cpp::OutOfFileDescriptorsException &e)
{
    cerr << "No more file descriptors" << endl;
    exit(1);
}
catch (oas::cpp::UnknownAddressException &e)
{
    cerr << "Address not known" << endl;
    exit(1);
}
catch (oas::cpp::ConnectionFailedException &e)
{
    cerr << "Connection failed. Try later" << endl;
    exit(1);
}
catch (oas::cpp::IOException &e)
{
    cerr << "IO Exception" << endl;
    exit(1);
}
catch (oas::cpp::InternalErrorException &e)
{
    cerr << "Internal error received: " << e.getMessage() << endl;
    exit(1);
}

catch (oas::cpp::Exception &e)
{
    cerr << "Unknown error received: " << e.getMessage() << endl;
    exit(1);
}
catch (...)
{
    cerr << "Unknown error obtaining root naming context" << endl;
    exit(1);
}

// Create a CosNaming name object to access the c++ cartridge instance
CosNaming::Name name;
name.length(2);
name[0].id = CORBA::string_dup("Bank");
name[1].id = CORBA::string_dup("Account");

try
{
    CORBA::Object_ptr cppobj_ptr = rootNC_var->resolve(name);

```

```

        if (CORBA::is_nil(cppobj_ptr))
        {
            cerr << "Error resolving Bank/Account" << endl;
            exit(1);
        }

        cout << "c++ object : " << cppobj_ptr << endl;
        Bank::Account_ptr account_ptr = Bank::Account::_narrow(cppobj_ptr);
        if (CORBA::is_nil(account_ptr))
        {
            cerr << "Error narrowing Account instance" << endl;
            exit(1);
        }

        //Get the smart var ptr for automatic release
        Bank::Account_var account_var = account_ptr;
        cout << "Balance: " << account_var->getBalance() << endl;
        account_var->deposit(100);
        cout << "Deposting: 100 bucks... " << endl;
        cout << "New balance: " << account_var->getBalance() << endl;
        account_var->destroy();
    }
    catch (CORBA::SystemException& se)
    {
        cerr << "Error: " << se._repository_id() << endl;
        return 1;
    }
    catch (...)
    {
        cerr << "Unknown error" << endl;
        return 1;
    }

    return 0;
}

```

---

# Installing C++ Applications

This chapter describes how to install and debug C++ applications in Oracle Application Server.

## Contents

- [Deploying Applications](#)
- [Creating Client Executables](#)
- [Using the Utilities](#)
- [Location of Your Registered C++ Application](#)
- [Reinstalling and Reloading Applications](#)
- [Configuring C++ Applications on Remote Nodes](#)
- [Debugging Applications](#)

## Deploying Applications

Deploying a C++ application on Oracle Application Server, consists of the following steps:

1. [Generating Stubs and Skeletons](#)
2. [Generating C++ Cartridge Factories](#)
3. [Creating the Shared Library](#)
4. [Installing the Application](#)

If you modify the code of your C++ application, you need to re-deploy the application so that the generated files are updated.

After you have installed the application, you can modify the values of its tuning parameters, if necessary, using the Oracle Application Server Manager. See *Oracle Application Server Administration Guide* for details. The default values of some configuration parameters are taken from the **CPP.app** deployment information file.

## Generating Stubs and Skeletons

You run the IDL C++ compiler to generate C++ stubs and skeletons for your IDL file. For example:

```
prompt> oasoidlc -g cplus -I$(ORACLE_HOME)/orb/4.0/include Bank.idl
```

This command generates two sets of files:

- **BankS.cpp, BankS.h**: The skeleton and its header for the interfaces in Bank.idl.
- **BankC.cpp, BankC.h**: The stub and its header for the interfaces in Bank.idl.

See [“The IDL C++ Compiler” on page 6-5](#) for more on using **oasoidlc**.

## Generating C++ Cartridge Factories

After creating skeletons and stubs, you create the instance factories for each of the cartridges in the application. You use the **cppgen** utility to create the cartridge instance factories. For example, the following command creates cartridge instance factories for the Bank application:

```
prompt> cppgen -a CPP.app -i Bank.idl -o .
```

This command generates two sets of files:

- **BankW.cpp, BankW.h**: The factory for each cartridge instance.
- **BankT.cpp**: The servant creation callback.

See [“The cppgen Utility” on page 6-9](#) for more on using **cppgen**.

## Creating the Shared Library

You create the shared library by linking in the object files of the generated code and the implementation object code.

### For Solaris

You need to dynamically link the file **libwrcc.so** to the cartridge shared library. The **libwrcc.so** file contains the C++ cartridge runtime. The path for this file is



**\$ORAWEB\_HOME/cpp/lib.** For example, for the Bank application the command will be as follows:

```
prompt>CC -G -o Bank.so BankS.o BankC.o BankW.o BankT.o AccountImpl.o -L
$ORAWEB_HOME/cpp/lib/ -lwrcc -lCstd -lCrun
```

---

---

**Note:** For Solaris, version 5.0 of the C++ compiler is required.

---

---

### For Windows NT

Follow these steps for compiling and linking C++ cartridges on NT:

1. Link the following libraries for creating the C++ cartridge DLL:
  - \$(ORACLE\_HOME)\orb\lib\yop40.lib
  - \$(ORACLE\_HOME)\orb\lib\yoc40.lib
  - \$(ORACLE\_HOME)\orb\lib\yock40.lib
  - \$(ORACLE\_HOME)\orb\lib\yosc40.lib
  - \$(ORACLE\_HOME)\orb\lib\yoi40.lib
  - \$(ORACLE\_HOME)\orb\lib\yu40.lib
  - \$(ORACLE\_HOME)\orb\lib\yot40.lib
  - \$(ORACLE\_HOME)\orb\lib\ydc40.lib
  - \$(ORACLE\_HOME)\orb\lib\yr40.lib
  - \$(ORACLE\_HOME)\orb\lib\ys40.lib
  - \$(ORACLE\_HOME)\orb\lib\yt40.lib
  - \$(ORACLE\_HOME)\orb\lib\yun40.lib
  - \$(ORAWEB\_HOME)\lib\libwrkcl.lib
  - \$(ORAWEB\_HOME)\lib\libwrcc.lib
2. For each cartridge DLL, create a DEF file. The DEF file should have the following entries:

```
EXPORTS
oracle_OAS_Cartridge_OASObject__getStubs
oracle_OAS_Cartridge_OASObject__getId
Bank_Account__getImpl
```

The first two entries in the exports section are required for every DLL. Then add one entry for each cartridge in the C++ application, which has the following format:

```
<applicationName>_<cartridgeName>__getImpl
```

If the application/cartridge name has '/'s in them, replace the forward slashes with '\_' character.

---

**Note:** The C++ compiler should support namespaces for compiling the stubs generated from compiling the IDL file.

---

## Installing the Application

You install the application shared library on the Oracle Application Server site by using **cppinstaller** utility. For example:

```
prompt> cppinstaller -a CPP.app -l Bank.so
```

This command registers your application with the Oracle Application Server Site Manager, and copies the application shared library to a standard location in the Oracle Application Server site. See [“The cppinstaller Utility” on page 6-9](#) for details.

You can also install your application using Oracle Application Server Manager. See the *Oracle Application Server Administration Guide*.

## Creating Client Executables

Creating a client executable consists of the following steps:

1. [Generating Stubs of the Standard CosNaming Interface](#)
2. [Compiling and Linking the Client Program](#)

### Generating Stubs of the Standard CosNaming Interface

You need to generate stubs of the standard OMG CosNaming interface **cosnam.idl**. For example:

```
prompt> oasoidlc -g cplus -A "oasoidlc.c-cplus-kwd=true" $ORACLE_HOME/orb/4.0/  
public/cosnam.idl
```

If the client is accessing a secure C++ cartridge, you need to use the SecCosNaming interface from the `$ORAWEB_HOME/public/secnaming.idl`. See chapter “Security for IIOP-Based Applications: EJB, ECO/Java, and C++” of the *Oracle Application Server Security Guide*.

See [“The IDL C++ Compiler” on page 6-5](#) for more on using `oasoidlc`.

## Compiling and Linking the Client Program

Now you need to compile and link the client program with the required libraries and runtime files. For example, to generate the client executable for the Bank application (on the Solaris platform) you need to give the following command:

```
prompt>CC -o client client.o ../server/BankC.o cosnamC.o $ORACLE_HOME/orb/4.0/  
lib/liborb.so $ORACLE_HOME/orb/4.0/lib/libyop.so $ORAWEB_HOME/cpp/lib/  
libcppcl.so -lthread -lsocket
```

The above command links the ORB runtime files (**liborb.so**, **libyop.so**), the client side library (**libcppcl.so**), the standard Solaris libraries (**lthread**, **lsocket**) with the client files to create the client executable.

Note that Oracle Application Server uses the client side library **libcppcl.so** to get the stringified IOR of the root naming context.

## Using the Utilities

This section gives details about how to use the following utilities:

- [The IDL C++ Compiler](#)
- [The cppgen Utility](#)
- [The cppinstaller Utility](#)

## The IDL C++ Compiler

The IDL to C++ compiler is `oasoidlc` and is located in the `$ORACLE_HOME/orb/4.0/bin` directory in your Oracle Application Server installed machine. The `oasoidlc` command accepts CORBA IDL as input and produces a variety of files that can be used to build Oracle ORB clients and servers.

OMG IDL specifications are accepted by the compiler. The IDL-to-C++ language mapping used by the compiler corresponds to the CORBA 2.2 specification.

The **oasoidlc** compiler also supports the pragmas ID, prefix, and version related to the CORBA 2.2 Interface Repository. See Chapter 6 of the CORBA specification located at <http://www.omg.org/> for more information.

### Example

The following command compiles the **simple.idl** file:

```
prompt> oasoidlc -g cplus simple.idl
```

This generates the client stub and server skeleton, and the interface definition files:

- **simpleS.cpp**, **simpleS.h**: The skeleton and its header for **simple.idl**.
- **simpleC.cpp**, **simple.h**: The stub and its header for **simple.idl**.

### Setting oasoidlc Environment Variables

The **oasoidlc** command provides certain environment variable settings, which can be set before executing the command. Each variable is identified by a name and may have one or more values.

For a UNIX c shell environment, the variable may be configured as follows using one of following ways:

- `oasoidlc.environment_variable`  
or, if the variable requires a value:  
`oasoidlc.environment_variable=value`

For example, the command **foo** uses the **foo.verbose** environment variable to print or not print any extra information when you invoke the command. The value of **foo.verbose** is set before executing **foo**, as demonstrated below:

- `% setenv foo.verbose=true`
- `environment_variable`  
Some environment variables are not associated with a command; thus, would not require the command name, **oasoidlc**, before the name. For example, **ys.log.msgFilePath** is not associated with any command, so it is set without the command as a prefix, as shown below:  
`% setenv ys.log.msgFilePath=/directory/for/log/file`

### Options

You can specify the options in any of the following ways:

- on the command line
- by environment variables
- as resource settings in your user environment (in **resources.ora** file)

Table 6–1 lists the options available with **oasoidlc**.

**Table 6–1** *oasoidlc compiler options*

Option	Description
-D <i>name</i> {= <i>val</i> }	Define a macro name. Without a value, equivalent to <b>#define <i>name</i> 1</b> . With a value, equivalent to <b>#define <i>name</i> <i>value</i></b> . All macro names defined in this manner are processed before any macros that appear in the input file.  Environment variable name: <b>oasoidlc.mnidlc.preprocess.define</b> .
-E	Run only the preprocessor on the input. The preprocessed output is written to stdout by default. Use <b>-o</b> to redirect the output to a file. Syntactic and semantic analysis are not performed, and no stubs are generated.  Environment variable name: <b>oasoidlc.mnidlc.preprocess-only=true</b> .
-g <i>language</i>	Specify the coding language for generating output files. The choices are <i>c</i> , <i>cplus</i> , and <i>java</i> .  Environment variable name: <b>oasoidlc.mnidlc.language</b> .
-h	Print usage information to stdout.  Environment variable name: <b>oasoidlc.mnidlc.show-usage</b> .
-I <i>pathname</i>	Search for <b>#include</b> files in the directory specified by <i>pathname</i> . See the following section titled <a href="#">Locating Include Files (-I)</a> for additional information.  Environment variable name: <b>oasoidlc.mnidlc.preprocess.include</b> .
-l	Generate headers only.  Environment variable name: <b>oasoidlc.mnidlc.header-only</b> .

**Table 6–1** *oasoidlc compiler options*

Option	Description
-n	Generate no output; just perform preprocessing, syntactic and semantic analysis. When used with -r, the IFR data file is still generated.  Environment variable name: <b>oasoidlc.mnidlc.no-output=true</b> .
-o <i>pathname</i>	Write generated files to <i>pathname</i> . When used with -E, <i>pathname</i> must specify a file name. Otherwise, it should be a directory.  Environment variable name: <b>oasoidlc.mnidlc.outputpath</b> .
-r	Save the repository contents after semantic analysis. The resulting data file can be loaded into the Interface Repository server.  Environment variable name: <b>oasoidlc.mnidlc.save-repository=true</b> .
-T	Print tracing information.  Environment variable name: <b>oasoidlc.mnidlc.verbose=true</b> .
-U <i>name</i>	Undefine a macro name. This is equivalent to <b>#undef <i>name</i></b> . All -U options are processed after the -D options and before any preprocessing is performed on the input file. Therefore, this option can only be used to undo macro definitions set using the -D option.  Environment variable name: <b>oasoidlc.mnidlc.preprocess.undef</b> .
-V	Print a version banner to stderr.  Environment variable name: <b>oasoidlc.mnidlc.print-version</b> .
-w	Only show the error messages.  Environment variable name: <b>oasoidlc.mnidlc.no-warn=true</b> .

### Locating Include Files (-I)

File names in **#include** statements are located in the following manner if they are not absolute file names:

- The directory containing the current file is searched if the filename is enclosed in double quotes (**#include "fn"**). The use of angle brackets (**#include <fn>**) suppresses the search in the current directory.
- If the include file is not located in the current directory, the directories specified with -I are searched in the order they appear on the command line.

## The cppgen Utility

You use the **cppgen** utility to create C++ cartridge factories. This utility is located in the **\$ORAWEB\_HOME/bin** directory in your Oracle Application Server installed machine. The syntax for **cppgen** is as follows:

```
prompt>cppgen -a <deploymentDescriptorFilename> -i <applicationIDLFilename>
```

You specify options on the command line. The available options are listed in Table 6–2.

**Table 6–2** *cppgen and cppinstaller options*

Option	Description
-a	Deployment descriptor filename. By convention, it is <b>CPP.app</b> .
-i	Application IDL file.
-l	Shared library/DLL that implements the C++ cartridge.
-o	Output directory where the generated files will be located.
-v	Verbose. Prints diagnostic information.

## The cppinstaller Utility

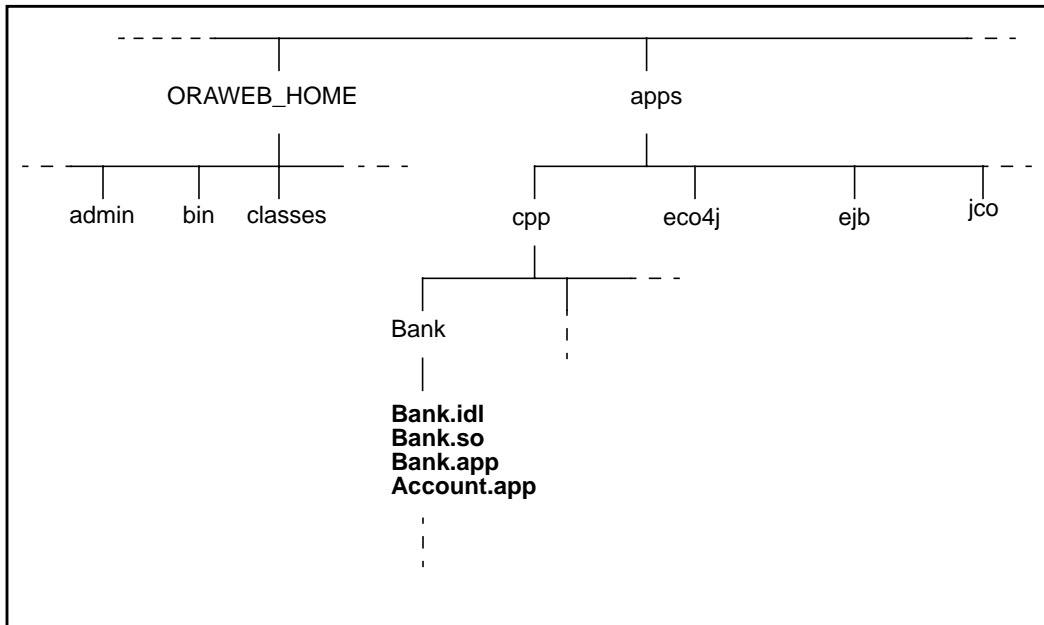
You use the **cppinstaller** utility to install your application shared library on the Oracle Application Server site. This utility is located in the **\$ORAWEB\_HOME/bin** directory in your Oracle Application Server installed machine. The syntax for **cppinstaller** is as follows:

```
prompt>cppgen -a <deploymentDescriptorFilename> -i <sharedLibraryName>
```

You specify options on the command line. The available options are listed in Table 6–2.

## Location of Your Registered C++ Application

Your registered C++ application is located in the **\$ORAWEB\_HOME/./apps/cpp/<appName>** directory. See [Figure 6–1](#).


**Figure 6–1 Location of a registered C++ application**

Note that **<appName>.app** is the master configuration file that is used to register the C++ application in Oracle Application Server. It includes the cartridge configuration files (the **<cartridgeName>.app** files).

## Reinstalling and Reloading Applications

If you modify your application (for example, if you modify the code, or you add or remove C++ cartridges to the application, or you modify the configuration parameters in the **CPP.app** file), you need to re-install the application using the Oracle Application Server Manager.

To re-install the application:

1. Stop any running processes of the C++ application that you need to re-install.
2. Delete the application.  
Select the application and click  .
3. Re-install the application.




4. Select “All” in the Oracle Application Server Manager and click the reload icon



so that the appropriate components of Oracle Application Server are notified. You do not have to re-start the application server.

If you change the configuration of the application using the Oracle Application Server Manager (for example, if you change the minimum number of instances or if you change the authentication string), you only need to reload the configuration data; you do not have to re-install the application.

To reload the configuration data, select “All” in the Oracle Application Server Manager and click the reload icon  so that the appropriate components of Oracle Application Server are notified. You do not have to re-start the application server.

---

**Note:** You cannot delete cartridges from a C++ application using the Oracle Application Server Manager, despite the fact that the delete button is active when you are viewing objects in the application.

To delete cartridges from a C++ application, you have to remove the cartridge and references to the cartridge from the deployment information file **CPP.app**, delete the application using Oracle Application Server Manager, and re-install the application.

---

## Reinstalling C++ Applications from the Command-Line

Instead of re-installing C++ applications using the Oracle Application Server Manager, you can re-install them using the **\$ORAWEB\_HOME/bin/cppinstaller** utility.

You need to set some environment variable before you can run **cppinstaller**. See Table 6–3. You also need to source the file **owsenv\_csh.sh**. See *Oracle Application Server Installation Guide* for details.

**Table 6–3** *Environment variables for the `cppinstaller` utility*

Environment variable	Value
ORAWEB_HOME	The top-level directory for Oracle Application Server. Example: <code>/private/app/oracle/product/8.0.4/ows/4.0</code> or <code>d:\orant\ows\4.0</code>
ORAWEB_SITE	The site name for Oracle Application Server. Example: <code>website40</code>
ORACLE_HOME	The directory that is the product base for Oracle products. Example: <code>/private/app/oracle/product/8.0.4</code> or <code>d:\orant</code>
CLASSPATH	The location of C++ class files that Oracle Application Server requires. Example:

Note that you can use `cppinstaller` to install C++ applications on the primary node only.

## Configuring C++ Applications on Remote Nodes

Follow these steps to configure a C++ application on a remote node:

1. Install the application on the primary node of the Oracle Application Server site. See [“Deploying Applications” on page 6-1](#).
2. Create the directory `$ORAWEB_HOME/../../apps/cpp/<applicationName>` on the remote node.
3. Copy all files from the directory `$ORAWEB_HOME/../../apps/cpp/<applicationName>` on the primary node to the directory `$ORAWEB_HOME/../../apps/cpp/<applicationName>` on the remote node.
4. In Oracle Application Server Manager, click `website40/Applications/<app-name>/Configuration/Server/Hosts`.
5. Click the newly added remote node for your application.
6. Reload Oracle Application Server.

## Debugging Applications

You should debug your C++ application as much as possible on your development platform before trying to debug it on Oracle Application Server's deployment platform. Debugging on a development platform is easier as you are working in a more controlled environment, which allows you to determine and fix problems more efficiently.

After you have tested and debugged C++ applications on your development platform, you deploy and test them on Oracle Application Server. Oracle Application Server provides debugging facilities in the form on logging messages.

### The Logger Class

The `oas::cpp::Logger` class enables C++ cartridges to access Oracle Application Server's logger service, which can write messages to a file or database. To get a logger object, you call the `getLogger()` method in the `oas::cpp::Context` class.

### Log Files

C++ cartridges can write messages to the log file that is used by Oracle Application Server, or they can write to a different log file.

- To write messages to Oracle Application Server's log file, you do not have to do anything. By default the logging service writes messages to the log file defined by the Logging Directory and the Log File fields in the System Logging form. To display this form in the Oracle Application Server Manager, click `website40/Oracle Application Server/Logging/System`.
- To write messages to a log file specifically for the C++ application, you specify the log file in the Logging form. To display this form in the Oracle Application Server Manager, click `website40/Applications/<appname>/Configuration/Logging`. The log file specified in this form is used only by the cartridges in this application.

You can also log messages to a database. See *Oracle Application Server Administration Guide* for information on database logging.

### Severity Levels

The Logger class writes messages only when they are at or below the severity level. For example, if you set the severity level of the application server to 7, only messages with severity level of 7 or lower are written to the log.

You set the severity level for Oracle Application Server as a whole in the Severity Level field in the System Logging form. To access this form in the Oracle Application Server Manager, click `website40/Oracle Application Server/Logging/System`.

You can override the overall severity level of Oracle Application Server for individual applications. You might want to do this if you want to see more messages only from specific applications. For example, you can set the overall severity level of Oracle Application Server to a low value, such as 1, but set the severity level of the application you are debugging to a high value, such as 10.

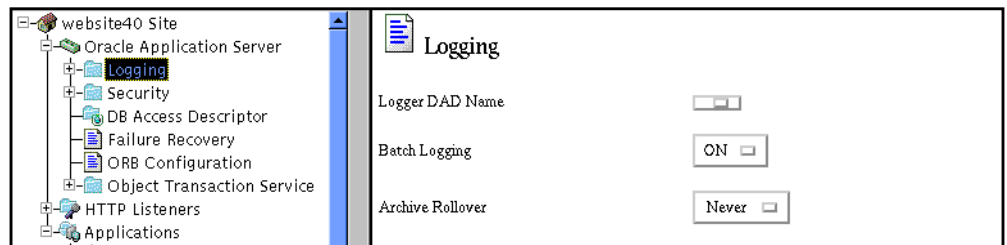
To override the severity level, you use the Logging form for the C++ application. To access this form, click `website40/Applications/<appname>/Configuration/Logging`.

To set the severity level of messages, call the `setMessageSeverity()` method in the `oas::cpp::Logger` instance. The severity level is then set for all messages sent from this C++ cartridge instance until you change it by calling the `setMessageSeverity()` method again. You can determine the current severity level by calling `getMessageSeverity()`. The default severity level is `Logger.LOG_SEVERITY_DEBUG`.

## Logging Modes

The logger can write messages as they occur or it can collect them and write them in batch mode. Batch mode is more efficient because the logger does not have to access the log file as many times as in non-batch mode. However, in batch mode, if the system fails, messages in the logger that are waiting to be written to the log file may not be written. Also, when you are debugging an application, you might want to set the batch mode to off so that you can see messages in the log file with minimal delay.

You set the batch mode in the Logging form ([Figure 6-2](#)).

**Figure 6–2 Logging form**

## Troubleshooting Tips

1. Increasing the logging level to 13 and checking the logging messages from the cartridge server.

You can use this tip to find out if your shared library has unresolved symbols.

2. Starting the cartridge server manually.

You can use this tip:

- If the cartridge server is core dumping. Starting the cartridge server manually will show the error messages on the screen.
- If you do not want to use the logger. Use the `printf()` function in the cartridge code, and start the cartridge server manually. You will be able to see the output of the `printf()` on the screen.

To start the cartridge server manually:

1. Source the **owsenv.sh** file.
2. Add **\$ORAWEB\_HOME/cpp/lib** to the **LD\_LIBRARY\_PATH**.
3. Use the command **wrks -s <appname>** to start the cartridge server.



This chapter contains reference pages for the following classes:

- [oas::cpp::Object Class \(Server Side\)](#)
  - [setContext](#)
  - [cppCreate](#)
  - [cppRemove](#)
- [oas::cpp::Context Class \(Server Side\)](#)
  - [getLogger](#)
  - [getUserTransaction](#)
  - [getParameters](#)
  - [remove](#)
  - [getObject](#)
- [oas::cpp::transaction::UserTransaction Class \(Server Side\)](#)
  - [begin](#)
  - [commit](#)
  - [rollback](#)
  - [getStatus](#)
  - [setTransactionTimeout](#)
  - [setRollbackOnly](#)
- [oas::cpp::Environment Class \(Server Side\)](#)

- 
- getNameByIndex
  - getParameterByName
  - getParameterByIndex
  - length
  - oas::cpp::Logger Class (Server Side)
    - << operator
    - setMessageSeverity
    - getLoggerSeverity
    - flush
  - oas::cpp::NSBootStrap Class (Client Side)
    - getOASRootNamingContext
    - freeIOR
  - oas::cpp::security::MessageDigest Class (Client Side)
    - getAlgorithm
    - getDigestLength
    - update
    - digest
    - digest
    - reset
  - oas::cpp::security::MessageDigestFactory Class (Client Side)
    - getInstance
  - Exception Classes
    - oas::cpp::CreateException
    - oas::cpp::RemoveException
    - oas::cpp::IllegalArgumentException
    - oas::cpp::IllegalArgumentException
    - oas::cpp::InternalErrorException
    - oas::cpp::UnknownAddressException

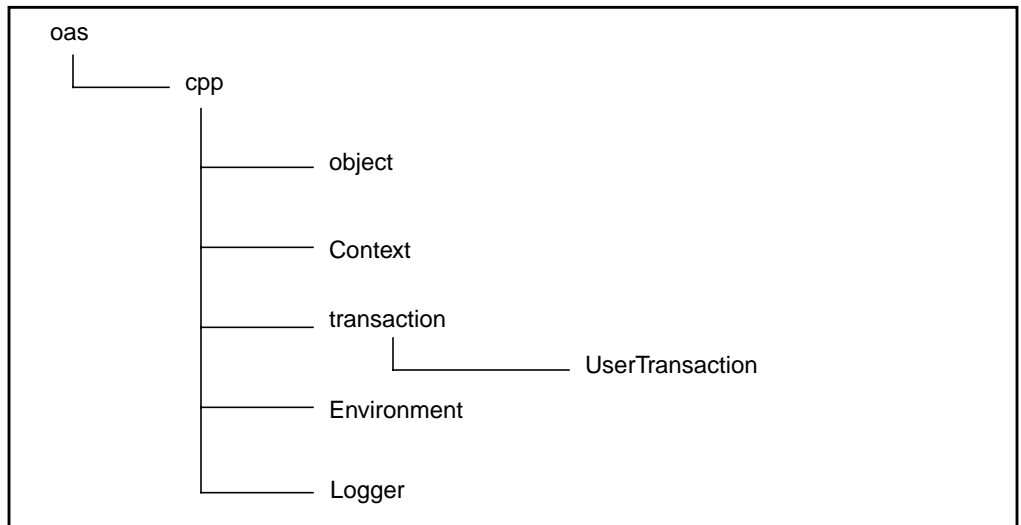


- `oas::cpp::URLFormatException`
- `oas::cpp::OutOfFileDescriptorsException`
- `oas::cpp::RootNCResolveFailedException`
- `oas::cpp::ConnectionFailedException`
- `oas::cpp::IOException`
- `oas::cpp::ListenerException`
- `oas::cpp::NameServerNotReachableException`
- `oas::cpp::security::NoSuchAlgorithmException`

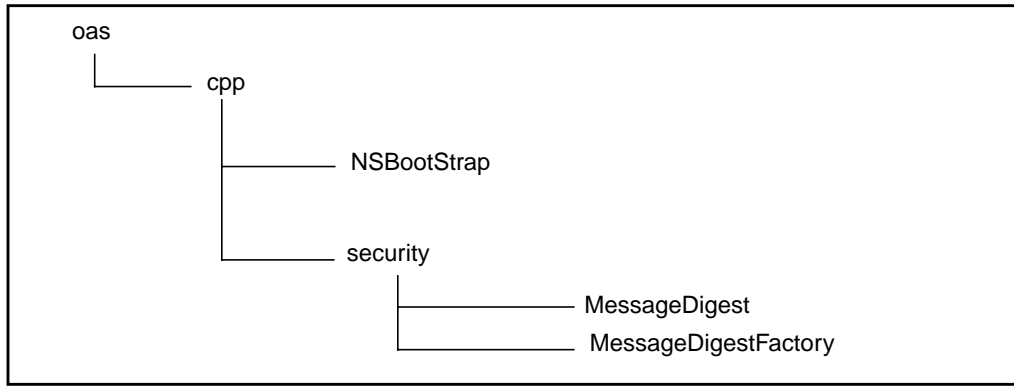
All the classes listed in this chapter are defined in the header file **cpp.h**. So, the C++ client code as well as the C++ cartridge code needing to use these classes should include the header file **cpp.h**.

All the above classes are arranged in a namespace hierarchy. Figure 7-1 and Figure 7-2 give a summary of all the server-side and client-side classes available to you for developing C++ applications.

**Figure 7-1 Server-side class heirarchy**



**Figure 7-2 Client-side class hierarchy**



---

## oas::cpp::Object Class (Server Side)

This class must be inherited by the C++ implementation class. It consists of the life cycle methods which are invoked by the container.

### setContext

#### Syntax

```
void setContext(oas::cpp::Context* ctx);
```

#### Description

This method is invoked by the container to provide the implementation object with the cartridge context. This method is invoked once by the container when the implementation object is created. It is invoked before `cppCreate()` method is invoked.

The cartridge context object (Context) is an important object that the implementation object uses to get the other objects like Logger, Environment, etc.

The implementation of the `setContext()` method should only store the Context object in an instance variable of the implementation class. It should not do other operation with the Context object.

The Context object should not be deleted by the implementation object.

#### Return Value

None

### cppCreate

#### Syntax

```
void cppCreate() throw CreateException;
```

## Description

Invoked by the C++ cartridge container when a new implementation object is created. The timing of the creation of a new implementation object depends on whether the cartridge is stateful or stateless.

An implementation object for the stateful C++ cartridge is created when a client does a `resolve()` on the cartridge node in the naming tree. An implementation object for the stateless C++ object is created when the client invokes a method call on a stateless C++ cartridge and no previously created implementation object is available to execute the method call.

## Exceptions

The implementation object can throw the `oas::cpp::CreateException` when there is an error in creating the implementation object. This exception is propagated to the client as the `CORBA::NO_RESOURCES` exception (for stateful C++ cartridges), or as `CORBA::NO_IMPLEMENT` exception (for stateless C++ cartridges).

For stateful C++ objects, this exception is thrown to the client when the client creates a C++ object using the `resolve()` or the `secure_resolve()` method on a cartridge node in the naming tree. For stateless C++ objects, this exception is thrown to the client when it makes a method call.

## cppRemove

### Syntax

```
void cppRemove();
```

### Description

This method is invoked by the container before the implementation object is deleted. The implementation object will be deleted either when it times out or after the implementation object called `remove()` on its Context object.

---

## oas::cpp::Context Class (Server Side)

This class provides a C++ implementation object with access to the C++ cartridge runtime container context. Each implementation object gets its own instance of the `oas::cpp::Context` class. The implementation object obtains this context through the `setContext()` method defined in the [oas::cpp::Object Class \(Server Side\)](#).

The implementation object should not delete the Context object.

### getLogger

#### Syntax

```
oas::cpp::Logger& getLogger(int severity = OASLogger::SEVERITY_DEFAULT);
```

#### Description

Each implementation object gets its own instance of the `oas::cpp::Logger` class. The implementation object can use the `getLogger()` method to get the Logger object.

The implementation object should not delete the Logger object.

#### Return Value

Reference to `oas::cpp::Logger` instance.

### getUserTransaction

#### Syntax

```
oas::cpp::UserTransaction* getUserTransaction();
```

**Description**

Gets the UserTransaction object. The C++ implementation object can use this to do transaction demarcation. The implementation object should not delete the UserTransaction object.

**Return Value**

The UserTransaction object.

## getParameters

**Syntax**

```
oas::cpp::Environment* getEnvironment();
```

**Description**

The implementation object uses this method to get its Environment object, through which it can access the environment parameters configured for the C++ cartridge. The implementation object should not delete the Environment object.

## remove

**Syntax**

```
void remove() throw (oas::cpp::RemoveException);
```

**Description**

Removes the C++ object. You invoke this method on the Context object when the client no longer needs the C++ object. This is required for both stateless and stateful C++ cartridges. Failure to do so can result in cartridge and memory leaks.

For stateful cartridges, calling remove() on the Context object will eventually result in the destruction of the implementation object. For stateless cartridges, calling remove() on the Context object will result in the destruction of the C++ object: the object reference of the C++ object, which was used to make this method call in the implementation class, will become invalid.

## Exceptions

Throws `oas::cpp::RemoveException` if an error occurs in destroying the C++ object.

## getObject

### Syntax

```
CORBA::Object* getObject();
```

### Description

Obtains the reference to the C++ object that was used to make this method call on the implementation class.

---

## oas::cpp::transaction::UserTransaction Class (Server Side)

You use `oas::cpp::transaction::UserTransaction` for transaction demarcation and setting transaction timeout values.

### begin

#### Syntax

```
void begin() throw (OASIllegalStateException);
```

#### Description

Begins a global transaction.

### commit

#### Syntax

```
void commit() throw (OASTransactionRolledbackException,  
                    OASHeuristicMixedException,  
                    OASHeuristicRollbackException,  
                    OASIllegalStateException);
```

#### Description

Commits an existing transaction. Transactions can only span method boundaries. If a transaction is not committed at the end of a method it is automatically rolled back.



# rollback

## Syntax

```
void rollback() throw (OASIllegalStateException);
```

## Description

Rollsback an existing transaction.

# getStatus

## Syntax

```
int getStatus();
```

## Description

Returns the transaction status. The valid values for the transaction status are listed in Table 7-1.

**Table 7-1** *UserTransaction constants*

Constant	Syntax	Description
STATUS_ACTIVE	static const int STATUS_ACTIVE	The transaction is associated with the target object and is in the active state.
STATUS_MARKED_ROLLBACK	static const int STATUS_MARKED_ROLLBACK	The transaction has been marked for rollback (perhaps as a result of setrollbackOnly operation).

**Table 7–1    *UserTransaction constants***

Constant	Syntax	Description
STATUS_NO_TRANSACTION	static const int STATUS_NO_TRANSACTION	The transaction is not associated with a target object.
STATUS_UNKNOWN	static const int STATUS_UNKNOWN	The transaction's status is unknown at this time.

## setTransactionTimeout

### Syntax

```
void setTransactionTimeout(long timeout);
```

### Description

Modifies the value of timeout that is associated with the transaction started by the current thread with the begin method.

## setRollbackOnly

### Syntax

```
void setRollbackOnly() throw (OASIllegalStateException);
```

### Description

Modifies the transaction associated with the current thread such that the only possible outcome of the transaction is to rollback.

---

## oas::cpp::Environment Class (Server Side)

This class provides a C++ implementation object with access to the C++ cartridge's environment parameters. These parameters are name-value pairs that can be configured in the deployment descriptor or through the Oracle Application Server Manager.

The cartridge can either look for a particular parameter name and get its value using the `getParameterByName()` method; or get all the configured parameters through an index using the `getParameterByIndex()` and `getNameByIndex()` methods.

### getNameByIndex

#### Syntax

```
const char *getNameByIndex(int index);
```

#### Description

Returns the name of the indexed environment parameter. If there are a total of N parameters, the index can range from 0 to N-1. If the index is out of range, this method returns a NULL. The returned name should not be modified.

### getParameterByIndex

#### Syntax

```
const char *getParameterByIndex(int index);
```

#### Description

Returns the value of the indexed environment parameter. If there are a total of N parameters, the index can range from 0 to N-1. If the index is out of range, this method returns a NULL. The returned value should not be modified.

## getParameterByName

### Syntax

```
const char *getParameterByName (const char *parameterName);
```

### Description

Returns the value of the given parameter name. If the parameter does not exist, this method returns NULL. The returned value should not be modified.

## length

### Syntax

```
int length();
```

### Description

Returns the number of environment parameters.

---

## oas::cpp::Logger Class (Server Side)

You use `OASLogger` to log the cartridge instance messages to the logging infrastructure. Messages can be configured to either be logged in a cartridge specific log file (one per process) or in the system-wide **wrb.log** file.

Severity level can be changed for each message logged. The Severity level is an integer ranging from 0 to 15. Level 0 is for fatal errors. Messages logged with level 0 cannot be prevented except by turning off system-wide logging. Level 15 is for least significant log messages.

The severity level with which the messages should be logged can be set either via cartridge configuration or via the `setMessageSeverity()` method. Whether the message is really logged or not depends on the system-wide log-level setting in the configuration.

### << operator

#### Syntax

```
OASLogger& operator<<(char arg);  
OASLogger& operator<<(char* arg);  
OASLogger& operator<<(const char* arg);  
OASLogger& operator<<(int arg);  
OASLogger& operator<<(void* arg);  
OASLogger& operator<<(long arg);
```

#### Description

You can use `oas::cpp::Logger` like “cout” to log different primitive data structures that include `char`, `char *`, `const char *`, `int`, `long` and `void *`. The `char *` and `const char *` should be null-terminated strings. The `void *` is printed as a pointer.

# setMessageSeverity

## Syntax

```
void setMessageSeverity(int severity = OASLogger::SEVERITY_DEFAULT);
```

## Description

Sets the message severity level. This will be the severity with which the subsequent messages will be logged.

## Parameters

*severity*: The severity level.

You can use the constants described in Table 7-2 to specify the severity of the message. The lower the value, the more severe the message is.

**Table 7-2** *Logger class constants*

Constant	Description
SEVERITY_FATAL	Severity of fatal error messages (1).
SEVERITY_WARNING	Severity of warning messages (4).
SEVERITY_INITTERM	Severity level is initiaize terminate (7).
SEVERITY_DEFAULT	The default severity level (11).
SEVERITY_TRACE	Severity of trace messages (15).

# getLoggerSeverity

## Syntax

```
int getLoggerSeverity();
```

## Description

Gets the current severity level for the logger which has been configured by the administrator. This can be used by the developer to make a decision as to whether a specific message should be logged or not.

For example, if the current severity level is 7 and you have a tracing message, you can ignore calling the write method since that message would not be logged.

**Return Value**

The current severity level.

**flush****Syntax**

```
void flush();
```

**Description**

Flush the stream written to by the user.

---

## oas::cpp::NSBootstrap Class (Client Side)

The C++ client can be local or remote to Oracle Application Server. Local clients are C++ applications or other cartridges running in the Oracle Application Server environment and can get direct access to the root naming context. Remote clients are outside Oracle Application Server and get access to the root naming context via a listener running in Oracle Application Server.

Various exceptions are thrown when the OASNSBootstrap object is used. All these exceptions derive from the OASException class.

### Example

Following is an example of remote client usage:

```
try
{
    const char*          rootNCobj_ior = NULL;
    CORBA::Object_ptr    rootNCobj_ptr = NULL;

    CosNaming::NamingContext_ptr rootNC_ptr = NULL;

    rootNCobj_ior = oas::cpp::NSBootstrap::getOASRootNamingContext ("oas://
www.oracle.com:80");
    rootNCobj_ptr = CORBA::ORB::string_to_object(rootNCobj_ior);
    oas::cpp::NSBootstrap::freeIOR (rootNCobj_ior);

    rootNC_ptr = CosNaming::NamingContext::_narrow(rootNCobj_ptr);
    if (CORBA::is_nil(rootNC_ptr))
    {
        cerr << "Error obtaining root naming context!" << endl;
        exit(1);
    }

    rootNC_var = rootNC_ptr;
}
```



## getOASRootNamingContext

### Syntax

For local clients:

```
const char *getOASRootNamingContext() throw (OASException);
```

For remote clients:

```
const char *getOASRootNamingContext(const char *url);
```

### Description

Returns the stringified IOR of the root naming context of the Oracle Application Server naming tree.

### Parameters

*url*: The URL in the form “oas::/host:port”. A trailing “/” will be ignored.

The “host” can either be the DNS name of the host, or the IP address in its standard “dot” format. This is the name of the machine on which Oracle Application Server listener is running. The “port” is the port on which the Oracle Application Server listener is listening.

### Exceptions

For local clients, throws:

- oas::cpp::NameServerNotReachableException
- oas::cpp::RootNCResolvedFailedException
- oas::cpp::InternalErrorException

For remote clients, throws:

- oas::cpp::URLFormatException
- oas::cpp::ListenerException
- oas::cpp::OutOfFileDescriptorsException
- oas::cpp::UnknownAddressException
- oas::cpp::ConnectionFailedException

- `oas::cpp::IOException`
- `oas::cpp::InternalErrorException`

## freeIOR

### Syntax

For local clients:

```
void freeIOR (const char *ior);
```

### Description

Frees the memory occupied by the stringified IOR returned by the `getOASRoot-NamingContext()` methods. Failure to call this method will result in a memory leak.

---

## oas::cpp::security::MessageDigest Class (Client Side)

This class encapsulates an engine that generates message digests from inputs of arbitrary length.

Since this is an abstract class, you need to create instances of this class by using the [oas::cpp::security::MessageDigestFactory Class \(Client Side\)](#) class.

See *Oracle Application Server Security Guide* for more information.

### Example

```
:
:
OASMessageDigest *md = 0;

try {

    // Create an instance of OASMessageDigest that implements the MD5
algorithm
    md = OASMessageDigestFactory::getInstance("MD5");

    ...

    md->update(buf,0,100); // update the engine with input

    unsigned char *out_buf;
    int dig_len = md->digest(&out_buf); // obtain digest value

    cout << "getAlgorithm=" << md->getAlgorithm() << endl;
    cout << "getDigestLength=" << md->getDigestLength() << endl;

    md->reset(); // reset the engine

}
catch ( ... ) {
    cout << "Exception caught" << endl;
}

...
```

## getAlgorithm

### Syntax

```
const char * getAlgorithm();
```

### Returns

The digest algorithm of this instance.

## getDigestLength

### Syntax

```
int getDigestLength();
```

### Returns

The length of the digest in bytes generated by this instance. This value is usually constant and does not depend on the length of input. For MD5, the digest length is 16 bytes (128 bit).

## update

### Syntax

```
void update(const unsigned char *input, int offset, int len) throw  
(OASIllegalArgumentException);
```

### Description

Updates the digest using the specified array of bytes.

### Parameters

*input*: The specified array of bytes to update the digest with.

*offset*: The offset to start from in the array.

*len*: The number of bytes to use.

### Exceptions

`oas::cpp::IllegalArgumentException`

## digest

### Syntax

```
int digest(unsigned char *buffer) throw (OASIllegalArgumentException);
```

### Description

Completes the digest computation and returns the digest as a new array.

### Parameters

*buffer*: A buffer allocated by the client.

### Exceptions

`oas::cpp::IllegalArgumentException`

## digest

### Syntax

```
int digest(unsigned char **buffer) throw (OASIllegalArgumentException);
```

### Description

Completes the digest computation and returns the digest as a new array. The number of bytes in the new array is returned.

### Parameters

*buffer*: The digest as an array of bytes. This array needs to be deleted by the client.

## Exceptions

`oas::cpp::IllegalArgumentException`

## reset

### Syntax

```
void reset();
```

### Description

Resets the digest engine.

---

## oas::cpp::security::MessageDigestFactory Class (Client Side)

You use this class to create instances of [oas::cpp::security::MessageDigest Class \(Client Side\)](#).

### getInstance

#### Syntax

```
static OASMessageDigest *getInstance(const char *alg) throw  
(OASNoSuchAlgorithmException);
```

#### Returns

Returns a MessageDigest instance that corresponds to the specified algorithm. The only supported algorithm is "MD5".

#### Parameters

*alg*: The specified digest algorithm. The only supported algorithm name is "MD5".

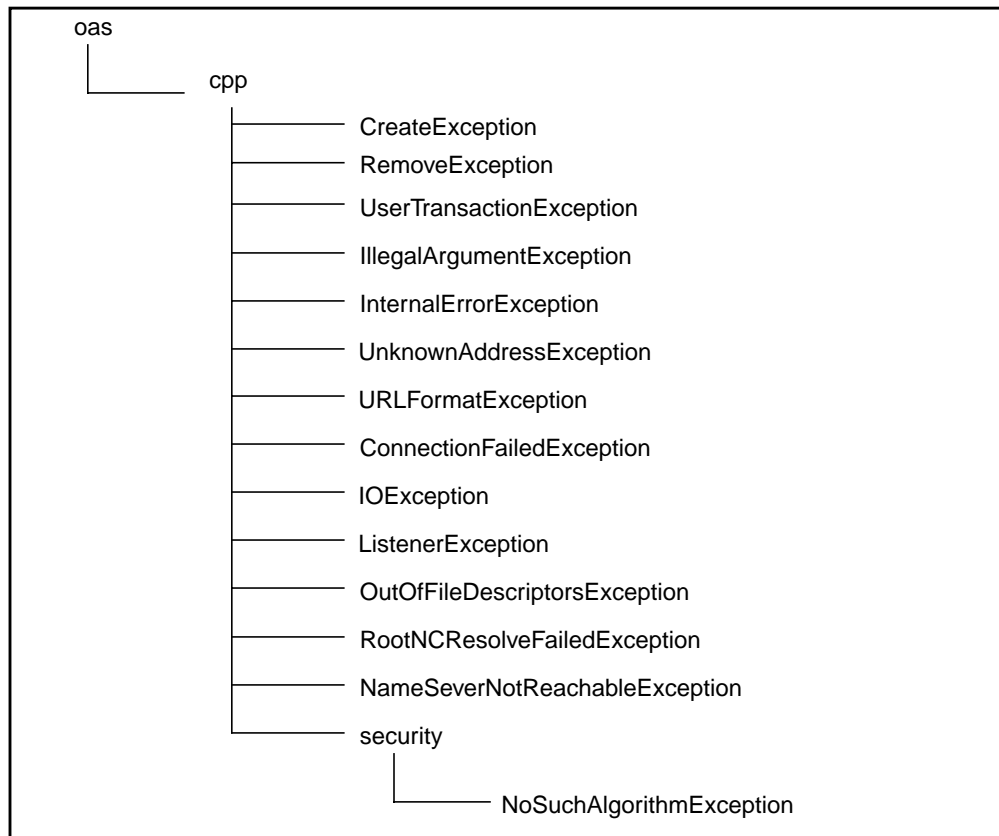
#### Exceptions

`oas::cpp::NoSuchAlgorithmException`

## Exception Classes

Figure 7–3 represents the class hierarchy of all the exception classes.

**Figure 7–3** *Class hierarchy of exception classes*



### **oas::cpp::CreateException**

Indicates that the creation of a C++ instance failed. The implementation object can throw this exception when the container calls `cppCreate()` on it. Implementation



objects should not throw this exception in their constructor or in the `setContext()` methods.

The `oas::cpp::CreateException` is propagated to the client as the `CORBA::NO_RESOURCES` exception (for stateful C++ cartridges), or as `CORBA::NO_IMPLEMENT` exception (for stateless C++ cartridges).

For stateful C++ objects, this exception is thrown to the client when the client creates a C++ object using the `resolve()` or the `secure_resolve()` method on a cartridge node in the naming tree. For stateless C++ objects, this exception is thrown to the client when it makes a method call.

## **`oas::cpp::RemoveException`**

Indicates that the remove of a C++ object failed.

## **`oas::cpp::IllegalArgumentException`**

Indicates that the input parameters are incorrect.

## **`oas::cpp::InternalErrorException`**

Indicates an internal error in Oracle Application Server. The message gives details of the internal error.

## **`oas::cpp::UnknownAddressException`**

Indicates that the given hostname cannot be resolved into an IP address.

## **oas::cpp::URLFormatException**

Indicates a format error in the URL. The message contains the part of the URL that has the format error.

## **oas::cpp::ConnectionFailedException**

Indicates that the connection could not be established with the Oracle Application Server listener. The most probable reason for this error is that the listener is not running.

## **oas::cpp::IOException**

Indicates an I/O error. This could be a network read/write error.

## **oas::cpp::ListenerException**

Indicates that the Oracle Application Server listener has not returned the IOR of the Oracle Application Server root naming context. The message contains the content returned by the listener.

## **oas::cpp::OutOfFileDescriptorsException**

Indicates that the process has run out of file descriptors.

## **oas::cpp::RootNCResolveFailedException**

Indicates that it is not possible to resolve the root naming context of the Oracle Application Server naming tree. The most probable cause is that the remote client did not specify the Oracle Application Server listener host:port as the URL.

## **oas::cpp::NameServerNotReachableException**

Indicates that it is not possible to resolve the root naming context of the Oracle Application Server naming tree. The most probable cause is that the naming server is not responding.

## **oas::cpp::security::NoSuchAlgorithmException**

Indicates that the specified algorithm is invalid.



## Symbols

---

<< operator, 7-15

## A

---

application development  
    general CORBA vs. C++ CORBA cartridge, 1-6  
application model  
    component-based, 1-1  
applications  
    C++ cartridge, 1-1  
        configuring C++, 6-1  
        creating, 3-1  
        installing C++, 6-1  
authenticationString property, 4-3, 4-4

## B

---

base class, 3-3  
begin() method, 7-10  
bootstrapping, 5-4

## C

---

C++ application  
    cartridges  
        creating, 2-2  
    configuring, 6-1  
    configuring on remote nodes, 6-12  
    creating, 3-1  
    creating remote interface, 3-1  
    definition, 1-2  
    deleting cartridges, 6-11

    deploying, 2-6  
    deployment descriptor file, 4-1  
        creating, 4-1  
    developing clients, 5-1  
    files required, 6-10  
    installing, 6-1  
    invoking methods on cartridges, 5-6  
    reinstalling, 6-10  
    reloading, 6-10  
    running, 2-12  
    tutorial, 2-1  
C++ cartridge  
    client view of, 1-3  
    creating, 2-2  
    creating remote interface, 3-1  
    definition, 1-2  
    deleting from application, 6-11  
    deploying the application, 2-6  
    implementation object, 3-2  
    overview, 1-1  
    runtime, 3-10  
    stateful vs.stateless, 3-10  
    transactions, 3-7  
    using, 5-6  
        using in an N-tier computing model, 1-7  
C++ client, 5-4  
    accessing the naming tree, 5-4  
    exceptions, 5-5  
C++ CORBA cartridge. *See* C++ cartridge  
C++ implementation object  
    definition, 1-2  
C++ object  
    definition, 1-2  
    destroying, 5-6

- getting object references, 5-2
- invoking methods, 5-6
- lifecycle, 1-5
- C++ object implementation, 1-3
- cartridge environment, 3-8
  - example, 3-9
- cartridge IDL file
  - creating, 2-2
- cartridge interface, 2-3
- cartridge server, 1-2
- cartridgeName.ENV section, 4-5
- classes
  - exceptions, 7-26
  - oas::cpp::Context, 7-7
  - oas::cpp::environment, 7-13
  - oas::cpp::Logger, 6-13, 7-15
  - oas::cpp::MessageDigestFactory, 7-25
  - oas::cpp::NSBootstrap, 7-18
  - oas::cpp::Object, 7-5
  - oas::cpp::security::MessageDigest, 7-21
  - oas::cpp::security::MessageDigestFactory, 7-25
  - oas::cpp::transaction::UserTransaction, 7-10
- client
  - bootstrapping, 5-4
  - C++, 5-4
  - client-side ORB, 5-2
  - creating, 2-8
  - Java, 5-5
  - security, 5-7
- commit() method, 7-10
- compiler, IDL C++, 6-5
- compiling IDL files, 6-5
- component-based application model, 1-1
- container
  - architecture, 1-4
  - definition, 1-2
- CORBA applications, 1-6
- CORBA::Object object, 5-6
- CORBA::TRANSIENT() exception, 5-6
- CosNaming, 5-4
- CPP.APP file, 2-5
- CPP.app file, 4-1, 6-2
- cppCreate() method, 7-5
- cppgen utility, 6-9
- cppinstaller utility, 6-9

- environment variables for, 6-12
- cppRemove() method, 7-6

## D

---

- deployment descriptor file
  - application section, 4-2
  - cartridge section, 4-3
  - creating, 4-1
  - name, 4-1
- deployment information file
  - creating, 2-5
- digest() method, 7-23

## E

---

- environment variables for cppinstaller, 6-12
- environment, cartridge, 3-8
  - example, 3-9
- exceptions, 7-26

## F

---

- files
  - CPP.APP, 2-5
  - CPP.app, 6-2
- flush() method, 7-17
- forms
  - Logging, 6-15
- freeIOR() method, 7-20

## G

---

- getAlgorithm() method, 7-22
- getDigestLength() method, 7-22
- getInstance() method, 7-25
- getLogger() method, 7-7
- getLoggerSeverity() method, 7-16
- getNameByIndex() method, 7-13
- getOASRootNamingContext() method, 7-19
- getObject() method, 7-9
- getParameterByIndex() method, 7-13
- getParameterByName() method, 7-14
- getParameters() method, 7-8
- getStatus() method, 7-11

getUserTransaction() method, 7-7

## H

---

home interface, 5-6

## I

---

### IDL

- compiling, 6-5
- language mapping, 6-5
- skeletons, generating, 6-6
- stubs, generating, 6-6

IDL C++ compiler, 6-5

implementation object, 3-2

implementationClass property, 4-4

implementationHeader property, 4-4

interface, cartridge, 2-3

## J

---

Java client, 5-5

JNDI, 5-5

## L

---

length() method, 7-14

load balancing, 1-5

log files

- C++ application, 6-13

logging, 3-4

- example, 3-5
- severity levels, 3-4

Logging form

- C++ application, 6-15

logging modes, 6-14

lookup() method, 5-6

## M

---

### methods

- begin(), 7-10
- commit(), 7-10
- cppCreate(), 7-5
- cppRemove(), 7-6
- digest(), 7-23

flush(), 7-17

freeIOR(), 7-20

getAlgorithm(), 7-22

getDigestLength(), 7-22

getInstance(), 7-25

getLogger(), 7-7

getLoggerSeverity(), 7-16

getNameByIndex(), 7-13

getOASRootNamingContext(), 7-19

getObject(), 7-9

getParameterByIndex(), 7-13

getParameterByName(), 7-14

getParameters(), 7-8

getStatus(), 7-11

getUserTransaction(), 7-7

length(), 7-14

lookup(), 5-6

remove(), 7-8

reset(), 7-24

rollback(), 7-11

setContext(), 7-5

setMessageSeverity(), 7-16

setRollbackOnly(), 7-12

setTransactionTimeout(), 7-12

update(), 7-22

mnidlc.language parameter, 6-7

mnidlc.no-output parameter, 6-7, 6-8

mnidlc.no-warn parameter, 6-8

mnidlc.outputpath parameter, 6-8

mnidlc.preprocess.define parameter, 6-7

mnidlc.preprocess.include parameter, 6-7

mnidlc.preprocess-only parameter, 6-7

mnidlc.preprocess.undef parameter, 6-8

mnidlc.print-version parameter, 6-8

mnidlc.save-repository parameter, 6-8

mnidlc.show-usage parameter, 6-7

mnidlc.verbose parameter, 6-8

## N

---

name property, 4-2

name value pairs, 3-8, 4-5

naming tree, 5-2

N-tier computing model, 1-7

## O

---

- oas::cpp::Context class, 7-7
- oas::cpp::Context object, 3-4
- oas::cpp::Environment class, 7-13
- oas::cpp::Logger class, 6-13, 7-15
- oas::cpp::MessageDigestFactory class, 7-25
- oas::cpp::NSBootstrap class, 7-18
- oas::cpp::Object class, 3-3, 7-5
- oas::cpp::security::MessageDigest class, 7-21
- oas::cpp::security::MessageDigestFactory class, 7-25
- oas::cpp::transaction::UserTransaction class, 7-10
- oasoidlc compiler, 6-5
- ORB
  - client side, 5-2
- overview
  - C++ cartridge, 1-1

## P

---

- parameters, environment, 3-8

## R

---

- remote interface
  - C++ application, 3-1
- remote node, configuring on, 6-12
- remoteInterface property, 4-4
- remove() method, 7-8
- reset() method, 7-24
- resource pooling, 1-6
- rollback() method, 7-11
- runtime, C++ cartridge, 3-10

## S

---

- SecNamingContext interface, 5-7
- security, 5-7
- setContext() method, 7-5
- setMessageSeverity() method, 7-16
- setRollbackOnly() method, 7-12
- setTransactionTimeout() method, 7-12
- severity levels, 3-4, 6-13
- stateful and stateless cartridges, 3-10
- stateless property, 4-4

## T

---

- terminology, 1-2
- timeout property, 4-3, 4-4
- transactionalDads property, 4-3
- transactions, 3-7
  - example, 3-7
- transactions property, 4-3
- troubleshooting tips, 6-15
- tutorial
  - C++ application, 2-1

## U

---

- update() method, 7-22
- utilities
  - cppgen, 6-9
  - cppinstaller, 6-9
  - oasoidlc, 6-5